Einführung in FORTH

aus:

Happy Computer -- PASCAL / FORTH / C
Sonderheft 5/1986

Markt & Technik

Programmiersprachen – Babel läßt grüßen

Wer den Computer zu seinem Hobby oder Beruf macht, steht in mehrerer Hinsicht vor einer schwierigen Wahl. Er muß sich zwischen einer Vielzahl von Peripherie, Betriebssystemen und Programmiersprachen entscheiden. Hier zeigen wir Ihnen die wichtigsten Programmiersprachen, ihre Entwicklungsgeschichte und was sie können.

Kommunikation braucht Sprache. Wenn sich zwei Menschen miteinander unterhalten, benutzen sie dazu das gesprochene Wort oder, sofern sie nicht die gleiche Sprache verstehen, Hände und Füße. Dieses Prinzip läßt sich nicht ohne weiteres auf die Verständigung zwischen Mensch und Computer anwenden. Kommunikation ist der Datenaustausch zwischen mehreren Parteien, die einander verstehen müssen, jedoch ist ein Computer im Urzustand alles andere als verständig. Er besitzt lediglich eine mehr oder weniger große Anzahl von Fähigkeiten, die er sehr schnell ausführen, aber nicht selbständig koordinieren kann.

Befehl und Gehorsam

Programmiersprachen sind, im Gegensatz zur weitverbreiteten Meinung, kein Mittel, um sich mit dem Computer zu unterhalten. Sie dienen lediglich dazu, dem Computer über eine Kette von Befehlen mitzuteilen, was er Schritt für Schritt zu tun hat. Programmierung wurde lange Zeit unter dem Hauptaugenmerk der Mensch-Maschine-Kommunikation betrachtet. Der wichtigste Aspekt lag darin, zu Problemlösungen unter möglichst effizienter Nutzung der Maschinenkapazität zu gelangen. Seit Computer in jüngster Zeit eine stärkere Verbreitung erfahren haben, rückte jedoch ein zweiter Gesichtspunkt immer mehr in den Vordergrund: Die Programme werden komplexer und der Wartungsaufwand (Korrektur oder Erweiterung eines Programms) immer größer. Dadurch, daß an vielen Programmen große Teams über lange Zeiträume hinweg arbeiten, geraten Programmiersprachen auch mehr und mehr zum Kommunikationsmittel zwischen diesen Gruppen von Menschen. Das bedeutet, daß ein guter Programmierer immer seine Programme auch für seine Nachwelt verständlich gestalten, und seine Kunstfertigkeit nicht mit der Anwendung von Programmiertricks unter Beweis stellen sollte. Einem potentiellen Benutzer, der das Programm lesen und eventuell ändern muß, sollte die Funktionsweise möglichst schon beim Lesen klar werden.

Statt »Programmiersprache« wäre die Bezeichnung »Kommandosequenz« eigentlich richtiger. Was dem Computer befohlen wird, führt er geduldig und beliebig oft aus, einzige Bedingung ist ein fehlerfreies Programm. Einige Psychologen behaupten denn auch, die Beliebtheit von Computern sei darauf zurückzuführen, daß viele Menschen ihre diktatorischen Triebe beim Programmieren ausleben!

Wenden wir uns der Frage zu, die wohl jeden Computeranwender irgendwann einmal bewegt, nämlich, was denn welche Programmiersprache wohl zu leisten vermag.

Im Laufe der Computergeschichte, die seit den ersten Relaisrechnern Konrad Zuses nicht mehr als ein halbes Jahrhundert zählt, wurde eine Unzahl Programmiersprachen, Spracherweiterungen und Dialekte entwickelt. Dabei standen immer zwei Überlegungen im Vordergrund. Zum einen mußten die Hardwarevoraussetzungen berücksichtigt werden. Zum anderen orientieren sich die Programmentwickler an den zu bearbeitenden Problemstellungen und den Bedürfnissen der Anwender. Je nach Computertyp und Problemstellung werden vom Programmierer verschiedene Programmstrukturen (Module, Blockkonzept, Verbundtypen), spezialisierte Befehle und unterschiedliche Datentypen (Integer, Real, Complex, String, etc.) benötigt. Auf dieser Grundlage entwickelten sich neben den bekannten Sprachen eine unüberschaubare Anzahl Exoten, die meistens nur in einzelnen Universitäten eingesetzt wurden.

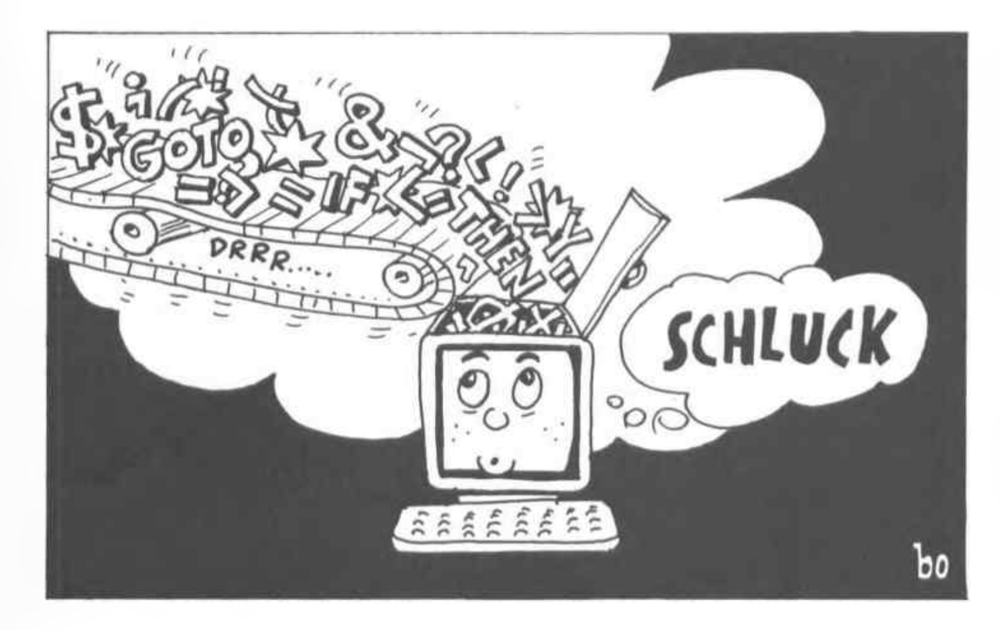
Vielfalt – Freud oder Leid?

An kaum einem Punkt scheiden sich die Geister in der Computerszene so stark wie in der Auswahl der Beurteilung der Programmiersprachen. Wer den Heimcomputer zu seinen Hobbies zählt, lernt in aller Regel zunächst fleißig Basic. Schließlich gehört ja der Basic-Interpreter zum Lieferumfang. Im Laufe der Zeit werden die eigenen Programme immer länger und unübersichtlicher, die einzelnen Programmteile sind durch ein unentwirrbares Geflecht von GOTO-Anweisungen miteinander verknotet (böse Zungen sprechen deshalb von »Spaghetti-Code«).

Mit dem wachsenden Bedürfnis nach Strukturierung, die in Basic nur jemand mit viel Selbstdisziplin erreicht, und nach Geschwindigkeit, die für Basic ein Fremdwort ist, sieht man sich nach Auswegen um. Dabei fühlt sich die eine Gruppe wie magisch von dem Begriff Assembler angezogen und begibt sich auf die unterste Sprachebene, um fortan in mühseliger Kleinarbeit Byte für Byte zu programmieren. Hiermit wird zwar ein Höchstmaß an Geschwindigkeit möglich, aber die Übersichtlichkeit kommt nach wie vor zu kurz. Die zweite Gruppe der Programmier-Gemeinde wendet sich deshalb modernen Hochsprachen zu, wie Pascal, Forth, Modula, Comal, Ada, C und so weiter. Diese Sprachen zwingen den Programmierer dazu, seine Programme modular (dies bedeutet, einzelne Aufgaben werden in »Paketen« oder »Modulen« zusammengefaßt) zu gestalten. Durch die so erreichte Übersichtlichkeit lassen sich Programme später von jedermann, Sprachkenntnisse vorausgesetzt, nachvollziehen und ändern. Zudem

sind einige dieser Sprachen sehr assemblernah, wie zum Beispiel Forth, womit auch Geschwindigkeit kein Problem mehr ist. Moderne Programmiersprachen unterstützen also Programmstrukturen und gehen ebenso mit Daten- und Kontrollstrukturen problemgerecht um.

Dennoch werden im professionellen Bereich (Universitäten, Verwaltung) »klassische« Sprachen bevorzugt, wie PL/1, Cobol und Fortran. Es sprechen auch gute Gründe dafür: So sind die neueren Programmiersprachen oftmals auf der vorhandenen Hardware noch nicht verfügbar, oder sie vertragen sich mit bereits vorhandenen Softwarekomponenten nicht. Ein anderer Faktor sind die Vorkenntnisse des Wartungspersonals und und und .... Jeder Informatikstudent, jeder Praktiker kann diese



Reihe beliebig fortsetzen. Und schließlich ist gute Software immer noch der Triumph des Programmierers, nicht der der Programmiersprache.

Seit das Betriebssystem CP/M auf Computern wie dem Commodore 128, Schneider CPC, oder dem Atari ST einen neuen Frühling erlebt, werden diese Klassiker neuerdings auch auf Heimcomputern interessant. Sehr wahrscheinlich wird sich ein breiter Anwenderkreis hierfür finden. Welcher Programmierer begrüßt es nicht, wenn er seine Produkte teilweise in der »guten Stube« austesten kann?

Eine Sonderstellung unter den Programmiersprachen nehmen die Sprachen für »künstliche Intelligenz« (KI) ein. Deren bekannteste Vertreter heißen Lisp und Prolog. KI-Sprachen bauen auf einem grundlegend neuen Konzept auf, bei dem der Programmierer dem Computer im Dialog sein Problem mitteilt. Wegen des hohen Speicherbedarfs sind KI-Programme auf Microcomputern nur sehr eingeschränkt einsatzfähig. Dies wird sich jedoch bald ändern. Man darf gespannt sein, wie sich KI auf den 16-Bit-Computern entwickeln wird, die ja nicht nur in punkto Schnelligkeit, sondern auch im Speicherangebot einen ganz neuen Standard setzen. Der Künstlichen Intelligenz ist in diesem Sonderheft ein eigener Beitrag gewidmet.

Fassen wir zusammen: Sinnvoll lassen sich die Programmiersprachen in vier Gruppen unterteilen:

1. Assemblersprachen: Sie bieten den Vorteil, daß sie die Möglichkeiten der Hardware optimal ausnutzen. Assembler versteht jeder Prozessor unmittelbar. Jede höhere Programmiersprache muß beim Programmablauf erst in Assemblercode übersetzt werden und ist deshalb weniger universell. Gegen Assembler sprechen die mühselige Programmierung, die Gebundenheit an Prozessor und Hardware sowie die schwierige Wartung.

2. Klassische Hochsprachen: Sie sind zur Zeit am weitesten verbreitet, aber genügen wegen ihrer altertümlichen Konzeption den Anforderungen moderner Programmierung nicht mehr.

3. Moderne Hochsprachen: Sie können sich gegen die »Alteingesessenen« nur langsam durchsetzen. Sie bieten Strukturierungshilfen, ausgeprägte Möglichkeiten der Datenbeschreibung, und unterstützen die Selbstdokumentation des Programmtextes durch eine angemessene Verbalisierung.

4. KI-Sprachen sind heutzutage noch Gegenstand intensiver Forschungen und stellen eine völlig neue und andersartige Klasse von Programmiersprachen dar. In diesem Beitrag wollen wir uns deshalb nicht weiter mit ihnen beschäftigen.

Wie Sie gesehen haben, ist der Sprachenwald immer noch sehr dicht, auch wenn man sich auf die Auswahl der wichtigsten Programmiersprachen beschränkt. Ein bedeutendes Auswahlkriterium sind die qualitativen Merkmale. Bevor wir diese besprechen, noch zu einigen zentralen Begriffen:

Algorithmus – Jedem Programm liegen ein oder mehrere Lösungsverfahren zugrunde, oft auch als Algorithmus bezeichnet. Ein solcher Algorithmus ist definiert als eindeutige und vollständige Vorschrift zur Lösung einer Problemklasse mit einer Abfolge von Schritten, die in einem endlichen Zeitraum ausgeführt werden.

Daten heißen dabei die Objekte, die der Algorithmus bearbeitet.

Programm nennt man folgerichtig die maschinengerechte Aufbereitung der Daten und des Algorithmus. Diese Aufbereitung geschieht mit Hilfe einer künstlichen Sprache, eben der **Programmiersprache**. Diese setzt sich aus einer Menge von Zeichen zusammen, die ihrerseits nach bestimmten Regeln zusammengesetzt werden können. Die somit geschaffenen Sprachelemente werden von der Maschine unmittelbar (als Maschinensprache) oder unter Zuhilfenahme von Übersetzungsprogrammen (als Hochsprache) »verstanden«. Der Übersetzer stellt nichts weiter dar, als einen speziellen Algorithmus, der in der Lage ist, alle Befehle einer Hochsprache in den entsprechenden Assemblercode zu transformieren. Er ist grob vergleichbar mit einer Bibliothek von kleinen Assembler-Unterprogrammen, wobei jedem Befehl der Hochsprache eines dieser Unterprogramme zugeordnet ist.

Compiler und Interpreter

Übersetzungsprogramme gliedern sich in zwei Typen, die die gleiche Aufgabe auf unterschiedliche Weise erfüllen. Als erstes sind die **Compiler** zu nennen. Sie tauchten auch in der geschichtlichen Entwicklung zuerst auf. Der Compiler übersetzt den Programmtext (Quellcode) in einem oder mehreren Durchgängen (Passes) komplett in Assemblercode (Objektcode). Der Compiler selbst wird daher beim Programmlauf nicht mehr benötigt. Natürlich benutzt man für unterschiedliche Computer, mit verschiedener Hardware und Maschinensprache auch unterschiedliche Compiler.

Der zweite im Bund der Dolmetscher nennt sich **Interpreter**. Er ist der fleißigere von beiden. Zunächst muß der Quellcode direkt im Arbeitsspeicher des Computers abgelegt werden. Sodann beginnt das Interpretieren. Das heißt nichts weniger, als daß der Interpreter während des Programmlaufs Befehl für Befehl holen muß. Natürlich ist diese Vorgehensweise in höchstem Maße unökonomisch und langsam. Während ein Compiler die ersten drei der genannten vier Arbeiten nur genau einmal ausführen muß, beschäftigen sie den Interpreter bei jeder Programmwiederholung aufs neue.

Zu Beginn des Computerzeitalters, als Rechenzeit noch sehr teuer war, wurden daher ausschließlich Compiler entwickelt. Interpreter konnten sich erst mit höheren Prozessorleistungen durchsetzen. Sie werden auch heute noch in Profikreisen wegen ihres gemächlichen Arbeitstempos verschmäht.

Die Qualität von Programmiersprachen

Um beurteilen zu können, welche Eigenschaften gute Programmiersprachen charakterisieren, wenden wir uns zunächst der Frage zu, welche Anforderungen an ein hochwertiges Programm zu stellen sind.

Die Forderung nach Fehlerfreiheit erscheint auf den ersten Blick banal. Die Erfahrung zeigt nämlich, daß 100%ig korrekte Programmsysteme ab einem gewissen Umfang kaum noch möglich sind. Hat ein Programm eine gewisse Komplexität erreicht, ist es fast unmöglich, seine Korrektheit zu beweisen, also Testdurchläufe zu finden, die tatsächlich alle Eventualitäten berücksichtigen. Als Maß für die Korrektheit eines Programmes wird deshalb häufig von der Zuverlässigkeit gesprochen. Diese gibt die Wahrscheinlichkeit an, mit der ein Programm für eine Zahl von Anwendungsfällen in einer bestimmten Zeitspanne fehlerfrei arbeitet. Gerade die jüngste Vergangenheit hat hierfür im Bereich der Mikrocomputer einige negative Beispiele geliefert. So konnten Fehler in einigen Betriebssystemen oftmals erst nach der Auslieferung der neuen Geräte beseitigt werden.

Verständlichkeit deckt sich mit Forderungen nach Lesbarkeit, Überschaubarkeit und Selbstdokumentation. Programme sollten sich, sobald sie eine gewisse Länge überschreiten, in funktionelle Einheiten gliedern. Andernfalls wird der Programmierer oft, noch während er an ein und demselben Programm arbeitet, an der selbst produzierten Unordnung scheitern. Man differenziert unter dem Aspekt der Verständlichkeit zwischen der statischen und der dynamischen Programmstruktur. Die statische Strukturgestaltung dient dem Ziel, ein übersichtliches Layout des Programmtextes zu gestalten. Hiermit wird der menschlichen Wahrnehmung Rechnung getragen, die sehr stark auf optischen Wahrnehmungen beruht. Ergänzend trägt die dynamische Strukturierung dazu bei, daß Programmabläufe unmittelbar aus dem Quelltext ersichtlich werden.

Ein Faktor der an diese Zusammenhänge anknüpft, ist die Änderbarkeit von Programmen. Wartungsfreundlichkeit bedeutet einerseits, daß Programme leicht an modifizierte Aufgabenstellungen angepaßt werden können. Andererseits spielt die Portabilität eine Rolle, wenn ein Programm zum Beispiel in einer neuen Softwareumgebung lauffähig gemacht werden soll.

Universalität sollte es einem guten Programm ebenfalls ermöglichen, ähnliche Aufgabenstellungen und Abwandlungen zu lösen.

Im Zusammenhang mit der Benutzerfreundlichkeit sollten Programme einen Dialog mit dem Benutzer ermöglichen und Eingabefehler abfangen, ohne falsche Ergebnisse oder gar Systemabstürze zu liefern.

Effizienz hat im Laufe der Entwicklung stark an Bedeutung verloren. Zu Zeiten, da Speicherplatz noch Mangelware war, galten die kürzesten Programme als die besten. Das ging natürlich zu Lasten der Übersichtlichkeit. Effizienzstreben wird heute daher nur noch mit Skepsis betrachtet.

Die hier genannten Qualitätsmerkmale stehen offensichtlich in positiver und negativer Wechselwirkung zueinander. So geht Übersichtlichkeit meistens zu Lasten der Effizienz, verbessert dagegen aber die Änderbarkeit.

Beginnen wir jetzt damit, die Anforderungen an eine ideale Programmiersprache zu formulieren. Die Qualität einer Computersprache läßt sich danach beurteilen, inwieweit sie die Entwicklung von Programmen unterstützt, die den uns bekannten Anforderungen entsprechen. Die folgenden skizzierten Merkmale müssen im engen Zusammenhang miteinander betrachtet werden.

Beginnen wir mit einem Punkt, der besonders den Einsteiger interessieren wird:

Die Erlernbarkeit einer Sprache hängt wesentlich von deren Struktur und Umfang ab. Sie ist sehr viel einprägsamer, wenn die Zahl der Schlüsselwörter gering und das Sprachkonzept durchgängig ist. Ebenso gewährleistet eine einfache Benutzung, wenn der Programmierer sein Problem bequem mit einer breiten Palette von Ausdrucksmöglichkeiten lösen kann.

Die Einheitlichkeit ist ein ebenso wichtiger wie unscharfer Begriff. Er soll im großen und ganzen bedeuten, daß für eine bestimmte Leistung möglichst nur genau ein Sprachmittel zur Verfügung steht.

Kriterien für eine ideale Sprache

Kompaktheit steht mit Einfachheit in Einklang. Hierbei ist nicht von der »Würze der Kürze« die Rede, sondern vielmehr von der Mächtigkeit der Sprachkonzepte. Der Bauplan der Sprache muß eine geringe Anzahl verschiedener Grundkonzeptionen aufweisen, wie mathematische Operationen, Ein- und Ausgabe, Datenstrukturierung. Andererseits soll Kompaktheit ein gewisses Maß an Redundanz (= alles was man in der gleichen Sprache auf andere Weise auch darstellen kann) an der richtigen Stelle nicht verhindern. Dies fördert die Verständlichkeit und Zuverlässigkeit. Beispielsweise sind Datentypen im Grunde genommen redundant, doch wer möchte sich schon mit einer Sprache herumschlagen, die ausschließlich den Datentyp »Zeichen« kennt?

Die Sprache Basic wird von vielen als katastrophal eingestuft. Das hat einen guten Grund: Die Forderung nach Lokalität wird von Basic so gut wie gar nicht unterstützt. Mit diesem Begriff ist gemeint, daß die Teile einer bearbeiteten Aufgabe, die logisch zusammengehören, auch im Programmtext in physischer Nachbarschaft stehen sollten. Die berühmt-berüchtigten Sprungbefehle bewirken jedoch das genaue Gegenteil.

Ein ganz anderes Kriterium ist die Sicherheit der Programmiersprache. Dazu zählt das Unterstützen der Fehlerfreiheit eines Programmes durch Sprachelemente. »On Error Goto« ist ein solcher weitverbreiteter Befehl. Des weiteren sind Sprachelemente zu nennen, die die Testphase des Programms fördern.

Der »Wildwuchs« der Implementierungen (= Anpassungen eines bestimmten Computers) bei Programmiersprachen, insbesondere bei den älteren, ist hinlänglich bekannt. Dialekte, Erweiterungen, aber auch Einschränkungen beeinträchtigen die Portabilität von Programmen im besonderen Maße. Mit der Standardisierung

Fortsetzung auf Seite 10

befassen sich daher nationale und internationale Institute. Die wichtigsten sind das Deutsche Institut für Normung (DIN), das American National Standards Institute (ANSI) und die International Organization for Standardization (ISO). Die Benutzer akzeptieren allerdings die Standards unterschiedlich. So sind Standards von Basic nahezu unbekannt, während sie sich bei Sprachen wie Pascal oder Fortran zunehmend durchsetzen.

Was für die Effizienz bei Programmen gesagt wurde, verkehrt sich bei den Sprachen in das genaue Gegenteil: Für den Übersetzer ist sie von großer Wichtigkeit. Zum einen sollte der Übersetzungsvorgang effizient sein, da bei der Fertigstellung eines Programmes die Zahl der Testläufe meistens sehr hoch ist. Der wichtigere Grund aber ist, daß das erzeugte Maschinenprogramm im Hinblick auf Rechenzeit und Speicherbedarf optimiert werden sollte. Sogenannte »Optimierende Übersetzer« sind teilweise in der Lage, das erzeugte Maschinenprogramm effizienter zu gestalten, als dies der Benutzer durch Programmiertricks erreichen kann.

Von Zuse bis Ada

Wir besitzen nun ein gutes Handwerkszeug, um Programmiersprachen nach den wichtigsten Gesichtspunkten theoretisch zu beurteilen. Fassen wir zusammen: Die entscheidenden Anforderungen an eine Sprache heißen Strukturierungshilfen, Selbstdokumentation, Datenbeschreibung, Benutzerfreundlichkeit und Zuverlässigkeit. Doch was nützt all die graue Theorie, wenn wir die Sprachen nicht kennen? Im folgenden werden daher die wichtigsten unter den bisher behandelten Aspekten und im Rahmen ihrer geschichtlichen Entwicklung vorgestellt.

Die Entstehung der Programmiersprachen orientiert sich immer auch an den Voraussetzungen der Hardware. Dies gilt insbesondere auch für die Gründerjahre.

Als geistiger Urvater der Rechenmaschinen mit Programmsteuerung darf der Engländer Charles Babbage gelten. Er begann 1833 mit der Konstruktion digitaler Rechenautomaten. Er legte seinen Maschinen aus Zahnrädern, Kurbeln und Hebeln zwei wichtige Erfindungen zugrunde, nämlich die Lochkartensteuerung und das Prinzip eines dekadischen Zählrades mit automatischem Zehnerübertrag. Babbages Projekte waren aber wegen fertigungstechnischer Schwierigkeiten nur in der Theorie funktionsfähig. Erst Elektromechanik und später die Elektronik machten die Rechenautomaten langsam zu Computern.

Der erste Rechenautomat der Welt mit Programmsteuerung wurde 1941 von Konrad Zuse in Betrieb genommen. Die Zuse Z3 war ein Relaisrechner, der bereits mit Dualzahlen arbeitete und zur Darstellung Gleitkommazahlen benutzte. Mit der Z3 waren neben den vier Grundrechenarten auch das Ziehen von Quadratwurzeln und das Potenzieren möglich. Ein Nachbau des historischen Modells, das im Krieg zerstört wurde, steht im Deutschen Museum in München. Der erste programmierbare Rechner Amerikas entstand 1944. Er wurde von dem Mathematiker Howard H. Aiken mit Unterstützung von IBM entwickelt und auf den Namen »Mark I« getauft. Er war jedoch ein Ungetüm von 16 m Länge und 35 Tonnen Gewicht und zudem langsamer als die früher entwickelte Z3.

Der Phase der Relaisrechner setzte der Einsatz von Elektronenröhren rasch ein Ende. Der bekannte ENIAC war die erste vollelektronische Rechenanlage der Welt und wurde 1945 in den USA fertiggestellt. Er erreichte gegenüber den Relaisrechnern bereits die 2000fache Rechengeschwindigkeit.

Während die Lochkarte nur eine starre Programmsteuerung ermöglichte (keine Schleifen, keine logischen Entscheidungen) begann man bald, sich über die flexible Speicherprogrammierung Gedanken zu machen. Als erstem gelang es dem Amerikaner John Neumann das genannte Problem auf einem Rechner zu verwirklichen. Der bereits 1944 von Neumann konzipierte Computer (EDVAC) erfüllte folgende Forderungen: Das Programm mußte, wie auch die zu verarbeitenden Daten, in der Maschine gespeichert werden. Außerdem benötigte man bedingte Befehle wie Vorwärts- und Rückwärtsverzweigungen. Jeden Befehl konnte zudem die Maschine selbst, wie jeden anderen Operanden ändern.

Befehle bestanden aus einem Operations- und einem Adreßteil. Im Operationsteil wird eine Angabe gemacht was zu tun ist (zum Beispiel Ausführung einer Multiplikation), der Adreßteil zeigt an, wo sich die zu verarbeitenden Daten befinden und wohin sie anschließend zu übertragen sind. Hier ist also die Rede von den ersten Assemblersprachen, an deren Grundprinzipien sich bis heute nicht geändert hat.

Der Schritt von der starren Programmsteuerung zum flexiblen Programm leitete die Wende vom Rechner zur Datenverarbeitung ein. Die Röhrencomputer der ersten Generation erreichten mit dem SSEC (Selective Sequenz Electronic Calculator) Ende der vierziger Jahre einen Höhepunkt. Dieser besaß nicht weniger als 12000 Elektronenröhren und etwa 21500 Relais und wurde von 36 Lochstreifenlesern gesteuert. Er führte die Berechnungen der Mondbahn durch, die 20 Jahre später im Apollo-Raumfahrtprogramm verwertet wurden. Mit dem Einzug der Transistortechnik und später mit den integrierten Schaltkreisen wuchsen fortan Rechnerleistung und Speicherkapazität immer schneller. Dies war die Voraussetzung für die Schaffung der höheren Programmiersprachen.

Die frühen Jahre – Fortran

Fortran ist die älteste der hier behandelten Hochsprachen und setzt einen Meilenstein in der Geschichte. Anfang der fünfziger Jahre wuchs die Zahl der Computer rasch. An Serienfertigung war noch nicht zu denken und es war jedes Gerät ein Einzelstück mit eigener Hardware und eigenem Assembler. So wurde bald der Wunsch nach einer Programmiersprache laut, die übertragbar und einfach zu programmieren sein sollte. 1952 wurde der Grundstein für Fortran gelegt, zu einer Zeit, da die Programmierung nur wenigen Spezialisten und ausschließlich in Assembler möglich war. John W. Backus war einer der Federführenden, dem die Programmiergemeinde Fortran zu verdanken hat.

Der Hauptgrund für die Entwicklung war die Schwerfälligkeit der Assemblerprogrammierung. 75 Prozent der Kosten eines Rechenzentrums verursachte damals die Fehlersuche. Verständlichkeit war daher ein wesentliches Entwurfsziel. Dadurch, daß die teuere Hardware optimal ausgenutzt werden mußte, waren die Rahmenbedingungen für Fortran bereits vorgezeichnet. Vorrangig wurden Sprachelemente implementiert, die der Speicher- und Laufzeiteffizienz nachkamen. Einige dieser Konzepte werden noch heute als sehr nachteilig angesehen – sind aber immer noch in Fortran enthalten.

1955 erschien ein Programmierhandbuch, und zwei Jahre später wurde die erste Implementierung auf einer IBM 704 freigegeben. Damit stand Fortran erstmals einer breiten Zahl von Programmierern zur Verfügung.

Der Name steht für FORmula TRANslating system (Formelübersetzer). Und genau dort liegt auch der Anwendungsschwerpunkt der Sprache. Rechnerische Probleme lassen sich in ihr leicht und natürlich ausdrücken. Damit wird

der Erlernbarkeit der Sprache Rechnung getragen. Im ingenieurwissenschaftlichen und mathematischen Bereich gilt Fortran auch heute noch als die wichtigste Programmiersprache. So bietet sie zum Beispiel neben den allgemein gebräuchlichen Zahlentypen Real (Fließkommazahlen) und Integer (Ganze Zahlen) auch noch den Typ »Double Precision« für Rechnungen mit höherer Genauigkeit sowie »Logical« für boolsche Operationen. Großrechner-Versionen beinhalten zudem noch den Typ »Complex«, der in der theoretischen Elektrotechnik eine sehr wichtige Rolle spielt.

Das Format dieser Sprache ist streng zeilenorientiert und erlaubt normalerweise nur einen Befehl je Zeile. Das hängt damit zusammen, daß Fortran zunächst als lochkartenorientierte Sprache entstand. Grundsätzlich mußte man damals für jede neue Anweisung eine neue Lochkarte (beziehungsweise Zeile) verwenden. Wie auch in Basic, das später aus Fortran entstand, mußte bei den ersten Versionen viel mit dem Goto-Befehl umhergesprungen werden. Neuere Versionen wie Fortran V und Fortran 77 bieten demgegenüber schon strukturierende Sprachelemente wie IF....THEN....ELSE...ENDIF.

Nachdem die 1958 geschaffene Version Fortran II eine mäßige Verbreitung gefunden hatte, entstand 1962 das in weiten Kreisen akzeptierte Fortran IV. Den fortschreitenden Auswüchsen immer neuer Versionen wurde 1966 Einhalt geboten, mit einer Version, die größtenteils mit Fortran IV identisch war. Schließlich überarbeitete das ANSI Fortran 66 im Jahr 1977 nochmals und beseitigte einige eklatante Mängel.

Unter CP/M ist Fortran derzeit für fast alle Mikrocomputer mit Z80-Prozessor erhältlich, ebenso wie eine Reihe von Fortran-Implementationen für MS-DOS-Computer.

Cobol – die Geschäftige

Die Programmiersprache Cobol entstand 1959 auf Initiative des US-Verteidigungsministeriums. Zu dieser Zeit begann Fortran sich gerade auszubreiten. Was noch fehlte, war eine Sprache für den kommerziellen und kaufmännischen Einsatz. So entwickelte man Cobol mit dem Ziel, große Datenbestände verarbeiten zu können und die Ein/-Ausgabe zu unterstützen. Insbesondere die ersten Fortran-Versionen waren hierfür ungeeignet. Ende der fünfziger Jahre wurde die Codasyl-Entwicklungsgruppe aus Vertretern der Computerindustrie und der amerikanischen Regierung gegründet. Schon 1960 stellte diese Gruppe die erste Version mit der Bezeichnung Cobol-60 vor. Sie war wesentlich an die weniger bekannte Sprache Comtran (COMmercial TRANslator) angelehnt. Aufgrund der hastigen Entwicklung von Cobol innerhalb eines halben Jahres ergaben sich viele Ungereimtheiten, die sich teilweise durch alle Neuentwürfe hindurchschleppten und auch heute noch nicht ganz beseitigt sind. Cobol-61 war dann die Grundlage für alle späteren Versionen. Sie war zu Cobol-60 nicht kompatibel. 1965 wurde als wesentliche Neuerung die Unterstützung von Massenspeichern und Tabellen mit eingebracht.

Die Sprachelemente von Cobol sind je nach ihrer Funktion in Module zusammengefaßt. Das ANSI entwickelte bis 1974 einen zwölf Module umfassenden Standard, namentlich Cobol ANS-74. Er wurde 1980 nochmals verbessert. Den jeweils neuesten Stand veröffentlicht das CODASYL-Komitee im Abstand von drei Jahren. Der Standard für die Bundesrepublik Deutschland ist in der DIN-Norm 66028 nachzulesen.

Die kurze Darstellung der Entwicklungsgeschichte läßt erkennen, daß Cobol ebenfalls eine alte Sprache ist. Cobol-Programme müssen aus heutiger Sicht als mangelhaft angesehen werden.

Ursprünglich verfolgte man wie bei keiner anderen Sprache das Ziel der Lesbarkeit des Programmtextes. Die Sprache sollte dann auch leicht erlernbar sein. Der Programmtext erinnert stark an (englische) Prosa. Cobol-Programme simulieren nämlich die natürliche englische Sprache. So wird zum Beispiel jeder Befehl mit einem Verb eingeleitet. Die Grundrechenarten stehen nicht als Symbole, sondern als Befehlswörter (add, divide) zur Verfügung. Ebenso werden logische Operatoren ausgeschrieben. Ein Beispiel: Wenn die Variable A größer ist als Null, soll der Variablen B die Summe der Variablen C und A zugeordnet werden. In Basic würde man das so formulieren:

```
IF A>0 THEN B= C+A
```

Daraus wird in Cobol:

```
IF A GREATER THAN ZERO ADD C TO
A GIVING B.
```

Daß so aus komplizierteren mathematischen Formeln monströse Gebilde werden, leuchtet ein. Da höhere mathematische Funktionen in Cobol ganz fehlen, ist die Sprache für wissenschaftliche Anwendungen völlig ungeeignet.

Derartige Beispiele verdeutlichen, daß das Entwicklungsziel von Cobol nicht sinnvoll erreicht wurde. Dennoch ist Cobol auch heute noch die weltweit am stärksten verbreitete Programmiersprache. Ganze Rechenzentren arbeiten mit ihr. Die hohen Investitionen und die Gewöhnung des Personals an diese Sprache wird auch in Zukunft für ihren Erhalt sorgen. Zudem erreicht auch noch keine neuere Programmiersprache die Cobol-Domäne Datenorganisation.

Wer einen Mikrocomputer mit den Betriebssystemen CP/M oder MS-DOS (PC-DOS) besitzt, kann in Cobol einsteigen.

PL/1 – von allem etwas

Fortran und Cobol sind charakteristisch für die strikte Trennung in kommerzielle und technisch-wissenschaftliche Anwendungen zu Beginn der sechziger Jahre. Daneben wurden Computer nur noch in Spezialgebieten eingesetzt. Im Laufe der Zeit traten aber die Merkmale der naturwissenschaftlich-technischen Bereiche in den betriebswirtschaftlichen Anwendungen immer mehr hervor und umgekehrt. So waren die kommerziellen Anwender mehr und mehr auf Methoden aus der Statistik, Operations Research und Ökonomie angewiesen. Mathematiker und Ingenieure stellten zunehmend höhere Ansprüche an Datenverwaltung und an die Ein-/Ausgabeunterstützung.

Als logische Konsequenz kamen die Anbieter den neuen Bedürfnissen mit einer universellen Hard- und Software-konfiguration nach. Hardwareseitig entwickelte IBM die Rechnerfamilie /360 mit dem Betriebssystem OS/360. Diese Anlagen zählten sich bereits zur dritten Computergeneration (das heißt, die Schaltkreise waren in Hybridtechnik ausgelegt, einer unmittelbaren Vorstufe der integrierten Schaltkreise).

Bei den Überlegungen zu einer neuen Sprache war man zunächst von Fortran ausgegangen. Die Organisation SHARE (Society for Help to Avert Redundant Effort), eine Vereinigung wissenschaftlicher IBM-Anwender, einigte sich mit der Firma IBM auf die Gründung eines Sprachkomitees. Zunächst war die Rede von Fortran VI. Man gelangte aber schon nach kurzer Zeit zu der Erkenntnis, daß die gewünschten Verbesserungen eine Kompatibilität mit Fortran unmöglich machten. Die Anlehnung an Fortran hätte außerdem die große Gruppe der kommerziellen Verwender abgeschreckt. So entschied man sich für die Entwicklung einer gänzlich neuen Sprache. Deren wichtigsten Entwurfsprinzipien waren: allgemeine Einsetzbarkeit und weitgehende Ausdrucksfreiheit. Der Sprachaufbau sollte modular sein und Testhilfen sowie Möglichkeiten zur

Fehlerbehandlung bieten. Ein großenteils gegenläufiges Ziel bestand in den Forderungen, den vollen Zugriff auf Hardware- und Betriebssystemleistungen bei gleichzeitiger Maschinenunabhängigkeit zu gewähren.

Nach vielen drastischen Überarbeitungen hatte sich der Sprachumfang bis 1965 stabilisiert. Nachdem die Abkürzung für NPL (New Programming Language) bereits vergeben war, einigte man sich schließlich auf PL/I (Programming Language one). Im August 1966 wurde dann der erste Compiler für eine IBM /360 freigegeben. Schließlich verabschiedeten das ANSI und die ECMA, ein europäisches Standardisierungskomitee, einen vorläufig endgültigen Standard. Die Verbreitung von PL/1 nahm zunächst rasch zu, wurde aber später den gesetzten Erwartungen nicht gerecht.

Kommen wir nun zu der Sprache, die jeder, und wenn nur vom Hörensagen, kennt. Sie kann sich rein zahlenmäßig im Mikrocomputerbereich als die am

In den Sprachumfang von PL/1 wurden viele Konzepte aus Fortran, Cobol, Algol und Jovial übernommen (letztere werden hier wegen ihrer geringen Verbreitung nicht behandelt). Leider gelang es nicht, sich bei der Auswahl nur auf die guten Eigenschaften der Vorgänger zu beschränken. Zudem ist der Sprachumfang riesig. Eine fast unüberschaubare Anzahl von Schlüsselwörtern zwingt zu Maßnahmen, durch die der Programmierer auch mit Teilmengen der Sprache sinnvoll arbeiten kann. Daher ergeben sich für die PL/1 -Syntax sehr freizügige Vorschriften. Einerseits existiert keine Zeilenstruktur, Zwischenräume, Einrücken und Kommentare dürfen fast beliebig verstreut werden. Da passiert es dann auch nicht selten, daß beim Programmieren das Format aus allen Fugen gerät. Außerdem sind die Schlüsselwörter nicht reserviert, wohl wegen ihrer großen Anzahl. Gebilde wie

```
IF ELSE=THEN THEN IF=ELSE; ELSE
IF=THEN
```

tragen wohl zur Verwirrung jedes hoffnungsvollen Programmierers bei. Ein wesentlich angenehmerer Fortschritt ist andererseits das ausgeprägte Blockkonzept der Sprache. PL/1 orientiert sich wesentlich an Prozeduren. Allgemein kann gesagt werden, daß die Einarbeitung in PL/1 viel Zeit braucht. Wer damit trotz aller Warnungen beginnen will, kann dies unter CP/M oder auf PC-Kompatiblen tun.

weitesten verbreitete Sprache der Welt rühmen. Sie ist schon fast als Teil der Allgemeinbildung zu betrachten. Ganz anders als bei den »großen professionellen« wurde die Entwicklung von Basic (Beginners All-purpose Symbolic



Instruction Code) nicht durch eine Industrie- oder Militärlobby getragen. Die Ausrichtung der Sprache kommt denn auch in ihrem Namen zum Ausdruck: Sie wendet sich an den Anfänger und soll für jeden Zweck geeignet sein.

Basic – Basis für Einsteiger

Ihre geschichtliche Entwicklung erklärt viele Aspekte des Sprachkonzepts. Basic wurde von Thomas Kurtz und John Kemeny von 1956 bis 1971 in den USA am Dartmouth College entwickelt. Ziel war es, Studenten, die sich nicht ausschließlich mit Ingenieurwissenschaften beschäftigten, das Programmieren zu erleichtern. So schlugen sich denn auch die Erfordernisse der allgemeinen Ausbildung einer Universität in der Sprache nieder: Bei der angesprochenen Zielgruppe erschien ein eigener Programmierkurs nicht erforderlich. Vielmehr sollte das Programmieren im Rahmen der Mathematikvorlesungen gelehrt werden. Hieraus erklärt sich auch die Ausrichtung von Basic auf mathematische Probleme. Die neue Sprache sollte leicht erlernbar und leicht zu benutzen sein. Dies war nach Meinung von Kurtz und Kemeny bei Fortran und Algol nicht der Fall. So erklärt sich auch, daß bei Basic nicht, wie bei PL/1, auf Bewährtes zurückgegriffen wurde.

Im Gegensatz zu den bisher behandelten Sprachen wurde Basic als vollständiges Programmiersystem konzipiert. Der Benutzer kann im Dialog mit Basic arbeiten, ohne die Basic-Umgebung zu verlassen. Hierzu existiert der »Direkt-Modus«, der das Editieren, Ausführen und Speichern von Programmen unterstützt. Basic ist zudem eine typische Interpreter-Sprache, für die allerdings auch verschiedene Compiler erhältlich sind.

Mit der Entwicklung des ersten Compilers begannen Kurtz, Kemny und eine Gruppe Studenten 1963. Im Mai 1964 wurde dann das erste Basic-Programm ausgeführt; die erste Version kannte nur 14 Instruktionen. Diese Minimalausstattung wurde am Dartmouth College bis 1971 in insgesamt sechs Versionen schrittweise vervollständigt. Seither nahmen die Autoren keine Veränderungen mehr vor. Das bedeutet natürlich nicht, daß Basic jemals eine wirkliche Standardisierung erfahren hätte. Im Gegenteil: Durch das Fehlen einer Interessenvertretung professioneller Anwender und durch die enorme Verbreitung der Sprache wucherten die Basic-Versionen fast uferlos. Für nahezu jeden Mini- und Mikrocomputer und selbst auf Taschenrechnern ist Basic erhältlich. Da von Herstellerseite nach dem Motto verfahren wurde »jedem Topf ein anderer Deckel« ist Kompatibilität ein Fremdwort. Auch die Bemühungen der ECMA und ANSI in den letzten Jahren waren kaum von Erfolg gekrönt. Lediglich ein »Minimal Basic« wurde kompromißbereit zum Standard deklariert. Diese Teilmenge ist der ersten Sprachbeschreibung von 1964 sehr ähnlich. Lediglich im Heimcomputerbereich gelang es mit den MSX-Computern erstmals, einen weitestgehenden Basic-Standard für Geräte verschiedener Hersteller zu schaffen. Haar in der Suppe war aber, daß die MSX-Computer wenig Verbreitung fanden.

Ein Basic-Programm besteht aus nummerierten Zeilen in aufsteigender Folge. Dabei sind die Zeilennummern aus einem festgelegten Intervall zu wählen. Jeder Zeilennummer folgen eine oder mehrere Instruktionen. Die

Zeilennummern legen die logische Reihenfolge der auszuführenden Anweisungen fest. Darüber hinaus dienen sie als Orientierung für die Sprungbefehle. Beim Editieren des Programmtextes lokalisieren sie die Zeilen. Basic ist noch stärker zeilenorientiert als Fortran. Das geht so weit, daß eine Basic-Anwendung durch die Zeilenlänge begrenzt wird. In einigen Basic-Versionen sind Fortsetzungszeilen mit dem Zeichen »&« möglich.

Als Datentypen sind in Basic nur numerische Daten und Zeichenketten vorgesehen. Eine Unterscheidung in ganzzahlig und Fließkomma, wie es von anderen Sprachen her bekannt war, wurde der Einfachheit wegen bewußt vermieden, ist aber dennoch auf vielen Computern implementiert. Für Datenstrukturen stehen nur ein- und mehrdimensionale Felder (Arrays) zur Verfügung. Deren Elemente können je nach Version entweder nur einzeln manipuliert werden, oder es stehen spezielle Matrizenoperationen bereit.

Die Sprunganweisungen sind, wie bereits erwähnt, vielen ein Greuel. Zwar bieten moderne Basic-Versionen viele Befehle, die das strukturierte Programmieren unterstützen, sie erfordern aber genaue Vorausplanung eines Programms. Im Regelfall endet bei längeren Programmen ein »Drauflosprogrammieren« im Chaos aus GOTO, GOSUB und FOR...NEXT.

Des weiteren beinhaltet Basic, je nach Ausstattung und Computer, Befehle für die Ein-/Ausgabe, mathematische Funktionen, Grafik, Tonerzeugung und den Zugriff auf das Betriebssystem. Hierauf im einzelnen einzugehen würde zu weit führen. Es sei daher auf entsprechende Fachliteratur verwiesen.

Pascal - strukturiert und einfach

Ein moderner Klassiker unter den Programmiersprachen ist Pascal. Der Name ist ausnahmsweise keine Abkürzung. Er wurde zu Ehren des französischen Mathematikers Blaise Pascal gewählt, der 1642 im Alter von 19 eine der ersten funktionsfähigen Rechenmaschinen konstruierte.

Die Wurzeln dieser Programmiersprache reichen in das Ende der sechziger Jahre zurück. Zu jener Zeit stellten Nikolaus Wirth und C.A.R. Hoare Überlegungen an, auf der Basis von Algol 60 eine Nachfolgesprache zu entwickeln. Algol 60 bot schon damals ein zufriedenstellendes Sprachkonzept. Das gilt besonders für die Strukturierung des Programmablaufs und des Programmtextes. Wegen dieser Vorteile wurde Algol zur Grundlage einer ganzen Klasse von Programmiersprachen, die bei Pascal beginnt und vorläufig von Ada gekrönt wird. Eine der Schwächen ist dagegen die unzureichende Datenstrukturierung. Ebenso wie Basic und Fortran kennt Algol 60 nur das Array. Die später folgende Version Algol 68 war ähnlich wie PL/1 zu umfangreich und unhandlich. Mit der Entwicklung von Pascal verfolgte man das entgegengesetzte Ziel. Der Schweizer Professor Nikolaus Wirth formulierte bei der Entwicklung der Sprache an der ETH Zürich die folgenden Schwerpunkte: Pascal sollte nur grundlegende Sprachkonzepte enthalten. Diese sollten natürlich definiert sein und das Erlernen des strukturierten Programmierens als eine systematische Disziplin unterstützen. Des weiteren sollte sie sich effizient auf allen Computern implementieren lassen.

Die erste vorläufige Version entstand 1968. Die vollständige Beschreibung eines Compilers und der Sprache selbst war 1971 fertig. Das 1974 erschienene Benutzerhandbuch »Pascal User Manual and Report« enthält eine Sprachdefinition, die heute als Wirth-Standard bezeichnet wird. Die verbreiteten Versionen Turbo- und UCSD-Pascal enthalten demgegenüber noch einige Erweiterungen, die vor allem Grafik und Zeichen-Strings betreffen.

Durch die leichte Erweiterbarkeit von Pascal entstanden bald viele Dialekte. Deshalb und wegen der weltweiten Anerkennung der Sprache nahmen sich verschiedene Normeninstitute diesem Problem an. Die ISO setzte schließlich 1980 einen Standard fest, der in der DIN-Norm 66256 nachzulesen ist. Strukturierung bedeutet nicht nur, daß das Programm übersichtlich ist, sondern daß sich der Vorgang des Programmierens in verschiedene Aktionen aufteilt. Bevor man sich an den Computer setzt, sollte man das Problem genau analysieren und in Aufgabenpakete zerlegen. Dann ist für jedes Paket ein Algorithmus zu bestimmen und in Pascal zu formulieren. Erst dann beginnt die Tipparbeit. Pascal-Programme entstehen also auf dem Papier und weniger am Bildschirm. Diese Vorgehensweise erreicht eine niedrige Fehlerquote und damit sinkt auch die Zahl der Übersetzerdurchläufe, was bei einer typischen Compilersprache wie Pascal sehr angenehm ist.

Ein Pascal-Programm ist klar gegliedert in einen Vereinbarungs- und einen Anweisungsteil. Vereinbart werden zuerst alle Variablen, Konstanten und deren Datentypen. Im Anweisungsteil werden die Aufgabenpakete in Procedures formuliert. Jede Procedure enthält einen eigenen Namen. Das eigentliche Hauptprogramm besteht dann nur noch aus dem Aufrufen der Procedures und steht im Programmtext ganz am Ende.

Daß der GOTO-Befehl in Pascal enthalten ist, verwundert eigentlich. Angesichts der Strukturbefehle IF....THEN-CASE kann man gut auf ihn verzichten. Der Vorrat an Datentypen eröffnet gegenüber Basic ganz neue Möglichkeiten. Man unterscheidet hier zwischen einfachen Typen, strukturierten Typen und Zeigertypen. Integer, Char, Boolean und Real zählen zu den einfachen Typen. Boolean bezeichnet eine Variable, der nur die Werte False oder True zugeordnet werden können. Array, Record, Set und File stehen als strukturierte Typen zur Verfügung. Set bezeichnet eine Menge. Auf diesen Typ sind die üblichen Mengenoperationen Vereinigung, Durchschnitt, Differenz, Untermenge und Elementüberprüfungen anwendbar. Record ermöglicht Verbundvariablen. Es lassen sich so unterschiedliche Variablentypen zu einer Variable zusammenfassen. Record war ursprünglich für kommerzielle Anwendungen gedacht (Tabellendarstellung). Demgegenüber werden mit dem Typ File nur Variablen eines einzigen Typs verkettet. Der Typ Zeiger (pointer) schließlich ermöglicht verkettete Listen und deren bequeme Manipulation, sowie Baumstrukturen. Wem das noch nicht reicht, der kann sich in Pascal weitere einfache Datentypen selbst definieren.

Pascal ist vielseitig und erzieht zum strukturierten Denken. Seine Verarbeitung, vor allem im akademischen Bereich, ist folgerichtig sehr hoch. So liegen denn auch für fast alle rechnereigenen Betriebssysteme wie auch für CP/M und MS-DOS Pascal-Versionen vor.

Forth – die etwas andere Sprache

Forth entstand Anfang der siebziger Jahre. Charles H. Moore entwickelte die Sprache ursprünglich zur Steuerung von Radioteleskopen. Er arbeitete dazu mit einem IBM-1130, einem Computer der dritten Generation. Das Endprodukt war aber so mächtig, daß es Moores Computer als einen der vierten Generation erscheinen ließ. Er wollte der neuen Sprache daher den Namen Fourth geben. Namen mit mehr als fünf Buchstaben waren auf dem IBM jedoch nicht erlaubt. So wurde das »u« ein Opfer dieser technischen Unzulänglichkeit.

Forth ist interaktiv wie Basic. Das heißt, es existiert sowohl ein Interpreter als auch ein Compiler. Programme können somit erst im Direktmodus »häppchenweise« getestet und anschließend compiliert werden. Des weiteren verbindet Forth Merkmale der Assemblersprachen mit denen der Hochsprachen.

Die Strukturierung in Forth entsteht durch die Definition immer neuer Worte. Der ohnehin schon große Sprachumfang nimmt beim Programmieren ständig zu. Beliebige Befehle können zu einem neuen Befehl zusammengefaßt werden, der dann sofort wieder in weitere Befehle mit eingebaut werden kann. Schließlich steht für das gesamte Programm ein einziger Befehl am Ende dieser Kette.

Selbstverständlich stellt Forth auch die Kontrollstrukturen zur Verfügung, die bereits für Pascal angesprochen wurden, wie IF.....ELSE....ENDIF, BEGIN...UNTIL, BEGIN...WHILE etc. Das berüchtigte GOTO fehlt hier ganz, ergäbe auch bei diesem Sprachkonzept keinen Sinn.

Grundlegendes Prinzip von Forth ist das Operieren mit dem Stapelspeicher (Stack). Dieser funktioniert nach dem LIFO-Prinzip (Last In, First Out). Alle Werte, die auf dem Stapel abgelegt wurden, lassen sich nur in umgekehrter Reihenfolge wieder herunternehmen. Für einen problemlosen Ablauf dieses Systems sorgen eine ganze Reihe von Stack-Befehlen, mit denen sich Werte verschieben und vertauschen lassen. Sämtliche mathematischen Operationen laufen in Forth über den Stack. Man bedient sich hierbei der Umgekehrten Polnischen Notation (UPN), die recht gewöhnungsbedürftig und Benutzern von HP-Taschenrechnern bekannt ist.

Forth ist ein sehr offenes System und durch seine Assemblernähe universell einsetzbar. Fehlende Funktionen können jederzeit selbst programmiert werden. Zudem ist Forth sehr schnell. Das Erlernen der Sprache und die Übersichtlichkeit der Programme können als noch ausreichend eingestuft werden. Auf jeden Fall fasziniert Forth jeden, der sich länger damit beschäftigt. Für alle verbreiteten Mikrocomputer existieren mittlerweile eine oder mehrere Versionen.

Logo — kinderleicht

Seymour Papert, der als geistiger Vater der Sprache Logo gilt, arbeitete 12 Jahre an der Verwirklichung dieser »Erziehungsphilosophie«. Er leitete ein eigens gegründetes Entwicklungsteam aus Programmierern und Lehrkräften am MIT (Massachusetts Institute of Technology) in den USA. Man arbeitete damals ausschließlich auf den größten vorhandenen Datenverarbeitungsanlagen. Dadurch fand ein wesentliches Konzept der KI-Sprache Lisp in Logo Anwendung: die Listenprogrammierung. Listen sind einfach zu definieren und können per Befehl manipuliert, kombiniert und verglichen werden. Eine leicht programmierbare Dateiverwaltung ist nur ein Anwendungsbeispiel dieser Technik.

Bekannt wurde Logo vor allem durch die Schildkröte, ein kleines Zeichensymbol der »Turtle-Graphics«. Mit ihr lassen sich auf einfache Weise die tollsten Grafiken zaubern. Die Schildkröte krabbelt über den Bildschirm und hinterläßt dabei eine sichtbare Spur.

Eine Schildkröte machte Logo bekannt

Gesteuert wird mit einfachen Befehlen wie FORWARD, BACK, LEFTTURN, RIGHTTURN. Zusätzlich muß noch die Länge der zurückgelegten Strecke und des Drehwinkels angegeben werden. Ebenso ist eine Standortabfrage der Turtle möglich. Logo-Programme ähneln in der Struktur dem Baukastenprinzip der Forth-Programme. Mit Hilfe des Interpreters lassen sich einzelne Bausteine erproben und später zum eigentlichen Programm zusammensetzen. Der Komfort ist dabei in Logo ungleich höher als in allen bisher genannten Sprachen. So kann der Anwender vorerst Begriffe wie Archivierung, Dateien und andere spezielle Funktionen der Datenverarbeitung links liegen lassen. Diese Vorteile gehen aber leider zu Lasten des Speicherplatzes.

Eng mit dem Prozedurkonzept verbunden ist die Rekursivität von Logo. Prozeduren sind in der Lage, sich selbst aufzurufen. Auf diese Weise lassen sich schnell reizvolle grafische Gebilde erzeugen und gewisse mathematische Zusammenhänge einfach ausdrücken. Rekursive Strukturen sind in Basic gar nicht und in vielen anderen Sprachen nur mit hohem Aufwand zu verwirklichen.

In die Logo-Philosophie wurden Erziehungstheorien des Schweizer Philosophen Jean Piaget eingebracht. Dieser hatte zuvor das Lernverhalten von Kindern analysiert. Tatsächlich wirkt Logo besser auf die Denkweise eines Schülers als Basic oder Pascal. Bemängelt werden muß bei Logo hauptsächlich die geringe Verarbeitungsgeschwindigkeit der Programme. Sie fällt aber bei einem Lernsystem nicht so stark ins Gewicht.

Wegen des hohen Speicherbedarfs sind Logo-Interpreter auf Mikrocomputern nur als Ausschnitt des Gesamtsystems erhältlich. Dies wird sich mit wachsendem Speicherstandard jedoch bald ändern.

Comal — gelungene Essenz

Im Jahre 1973 ging Comal aus den Sprachen Basic und Pascal als ein neuer Ableger hervor. Später kamen in Comal noch Elemente von Logo hinzu, so zum Beispiel die Schildkrötengrafik. Zudem sind in Comal der Compiler und der Interpreter nicht getrennt vorhanden, sondern es wurden deren beste Elemente in einer Zwischenstufe zusammengefaßt. Ein Comal-Programm besteht aus drei Schritten. Im ersten wird schon bei der Programmeingabe die Syntax überprüft. Dieser Syntaxchecker ist selbst für den ohnehin schon eingabefreundlichen Interpreter ungewöhnlich komfortabel. Die Comal-Schlüsselworte werden sofort in sogenannte »Token« übersetzt, das sind Abkürzungen, die nur ein Byte beanspruchen. Dieses Prinzip verwenden übrigens alle Interpreter. Der zweite Schritt beginnt nach dem Programmstart. In einer Art Compilerdurchlauf wird der Programmtext nach Variablen, Prozeduren, Funktionen und Sprüngen durchsucht. Die Ergebnisse dieser Analyse werden dann in einer gesonderten Liste zusammengefaßt. Man kann diesen Vorgang auch als eine Art automatische Deklaration ansehen. Im dritten Schritt, dem Programmlauf selbst, wird auf diese Liste ständig direkt zugegriffen. So ergeben sich gegenüber Basic, wo der Interpreter oft den ganzen Programmtext absucht, enorme Geschwindigkeitsvorteile.

Die Comal-Syntax lehnt sich stark an die von Basic an. Im Direktmodus wurden die meisten Befehle Wort für Wort übernommen. Das gleiche gilt für einige Befehle des Programm-Modus. Einige Comal-Versionen akzeptieren neben den eigenen Schlüsselworten sogar noch gleichbedeutende Basic-Befehle. Wer aus Basic zu Comal aufsteigt, ist so zwar vor Irrtümern einigermaßen sicher, jedoch wird dem Prinzip der Eindeutigkeit nicht gerade Rechnung getragen.

Von Pascal wurde die Strukturiertheit übernommen, dessen strenge Syntaxvorschriften aber erfreulicherweise vermieden. Die typischen Kontrollstrukturen, die schon bei Pascal und Basic beschrieben wurden, sind ausnahmslos vorhanden. Die Lesbarkeit des Pro-

grammtextes wird zudem dadurch unterstützt, daß Comal beim Listen eine optische Gliederung vornimmt.

Comal wird für fast alle gängigen Mikrocomputer angeboten. Erfreulicherweise stehen auch, ähnlich wie bei Forth, zahlreiche Public-Domain-Versionen zur Verfügung. Diese unterscheiden sich von kommerziellen Sprachangeboten lediglich im Umfang. Es wird erwartet, daß Comal in Zukunft eine starke Verbreitung erfährt. Jüngster Anhaltspunkt für diese These ist der Beschluß der Kultusministerien, Pascal im Informatikunterricht allmählich durch Comal zu ersetzen.

C — die Zukunft?

»C« - für diesen einen Buchstaben lassen viele Programmierer Pascal, Forth oder Fortran links liegen. Viele betrachten C als die Programmiersprache schlechthin. Tatsächlich bietet C viele bestechende Vorteile, die sich von denen der anderen Hochsprachen unterscheiden. In C wurden bekannte Betriebssysteme wie Unix und GEM geschrieben.

Die Entwicklung von C reicht bis an den Anfang der siebziger Jahre zurück. Die Namensgebung war ebenso kurios wie einfallslos. 1970 begann die Firma Digital Equipments mit der Entwicklung von Spezialsprachen, die die Programmierung von Minicomputern unterstützen sollten. Diese wurden einfach nach dem Alphabet benannt. B war also eine Weiterentwicklung von A und diente 1971 dazu, Unix auf verschiedene Rechner zu übertragen. Bald darauf erkannten Dennis Ritchie und Ken Thompson, die damals bei Bell Laboratories beschäftigt waren, die Leistungsfähigkeit von B. Sie verbesserten diese Sprache bis 1973 entscheidend. Wie das Endprodukt heißt, wissen wir bereits. Unix wurde wenig später ebenfalls auf C umgeschrieben.

Wenn ganze Betriebssysteme in C gehalten sind, läßt sich die ungeheuere Effizienz dieser Sprache schon erahnen. Das C-Compilat geht mit dem Speicher äußerst sparsam um, und ist zudem um den Faktor 50 schneller als vergleichbare Basic-Programme.

Der eigentliche Befehlsvorrat von C ist denkbar klein. Er umfaßt nur 13 Instruktionen. Bei der Erzeugung eines C-Programms wird der Quelltext zuerst mit dem mitgelieferten Editor oder einer Textverarbeitung geschrieben. Dieser dient dann als Eingabedatei für den Compiler. Bis hierher besteht also kein Unterschied zur Vorgehensweise mit den meisten anderen Compilersprachen. Der Compiler überprüft den Text und übersetzt diesen mit den ihm bekannten Befehlen in eine Zwischendatei und legt diese auf einen externen Speicher (zum Beispiel Diskette) ab. Anschließend beginnt das sogenannte »Linking«. Der Linker ordnet aus einer externen Bibliothek die noch fehlenden Befehle entsprechendem Assemblercode zu. So entsteht schließlich das lauffähige Programm, das fast so kompakt ist, als wäre es direkt in Maschinensprache geschrieben. Das Compilieren benötigt jedoch wegen des Zwischencodes je nach Geschwindigkeit des Speichermediums mehr Zeit, als die unmittelbare Übersetzung in Maschinensprache.

Die Vorteile des Compiler-Linker-Prinzips wiegen diesen Nachteil bei weitem auf: C ist praktisch uneingeschränkt portabel und die Linker-Funktion kann man selbst erweitern. C-Programme bieten neben der selbstverständlichen Strukturierung einige ungewöhnliche aber vorteilhafte Eigenschaften. So sind in Anlehnung an Assembler Befehle zum Inkrementieren und Dekrementieren vorhanden, Variable können als Registervariable deklariert werden. Daß die Benutzung derartiger Sprachmittel einem Compiler das Leben leicht macht, ist klar. Einschränkungen entstehen jedoch bei Prozessoren, die nur wenige Register aufweisen, wie beispielsweise die drei 8-Bit-Register des 6502, der nur A, X und Y zu bieten hat.

Ein weiteres nennenswertes Stilmittel unter C sind Makros. Diese sind mit Unterprogrammen vergleichbar, die nicht angesprungen werden müssen, sondern jeweils erneut vom Compiler in den Objektcode eingebunden werden. Diese Technik bringt einen großen Zeitvorteil, da Parameterübergaben und Sprünge entfallen.

Im 8-Bit-Bereich ist C nur auf dem Z80 unter CP/M verbreitet. Eine Version für den C 64 stellt hier eine erfreuliche Ausnahme dar. Für 16- und 32-Bitter existieren aber umfangreichere Versionen.

Ada — gekrönter Adel

Ada ist heute als Krönung bei der Entwicklung modularer Programmiersprachen anzusehen. Die Sprache wurde erst in jüngster Zeit im Auftrag des weltweit größten Softwaresponsors entwickelt, dem Pentagon. Der finanzielle und organisatorische Aufwand dafür war entsprechend riesig. Benannt ist die Sprache nach der jungen Gräfin Ada Byron, die um 1830 für Babbages (siehe oben) Rechenmaschinen ein nahezu komplettes Programm zur Berechnung der Bernoullischen Zahlen schrieb.

Der Aufwand, der um Ada getrieben wurde, erklärt sich mit einer Kalkulation des US-Verteidigungsministeriums. Danach können zwischen 1983 und 1999 etwa 24 Milliarden Dollar (!) eingespart werden, wenn eine einzige universelle Programmiersprache die bisherigen 450 (!!) Programmiersprachen ersetzen könnte. Ada ist ähnlich PL/1 sehr umfangreich. Es wäre daher zwecklos, auf einzelne Sprachelemente einzugehen. Deshalb hier nur die grundsätzlichen Sprachkonzepte von Ada:

- Das Modulkonzept von Ada ist äußerst umfangreich. Es stehen sowohl datenorientierte als auch funktionsorientierte Module zur Verfügung. Innerhalb der datenorientierten Pakete lassen sich fast beliebige Datentypen und Datenstrukturen realisieren.
- Ähnlich wie in Pascal können Datenstrukturen geschachtelt werden. Umfangreichere Prozedur- und Funktionskörper werden ausgelagert, zum Beispiel um die Lesbarkeit der Programme zu erhöhen.
- Sämtliche Kontrollstrukturen, von UNTIL bis hin zu CYCLE-Schleifen stehen zur Verfügung. Ferner sind alle linearen und strukturierten Datentypen implementiert.
- Ein automatischer Textformatierer (Pretty-Printer) wertet Schachtelungsstrukturen aus und sorgt für ein übersichtliches Layout des Programmtextes.
- Neben Parallelverarbeitung (Multitasking) gehören Konzepte wie die Parallel- und Ausnahmebehandlung zu den bemerkenswerten Fähigkeiten, auf deren Erklärung hier wegen der komplexen Zusammenhänge verzichtet wird.

Im großen und ganzen wurden die gesetzten Ziele bei der Entwicklung von Ada nach dem heutigen Erkenntnisstand optimal erreicht. Die zu Anfang unter den Qualitätsaspekten genannten Stichworte wie Vollständigkeit, Zuverlässigkeit, Korrektheit, Übertragbarkeit, Wartung und Fehlerbehandlung wurden weitestgehend realisiert. Die Einfachheit der Sprache ist als ausreichend zu betrachten, wenn auch die vollständige Einarbeitung in dieses System Jahre beansprucht. Ada ist für Mikrocomputer zur Zeit nur unter MS-DOS verfügbar und auch hier nur in einer abgespeckten Version.

Wir sind nun mit dem Aufstieg im »modernen Turm zu Babel« fast an der Spitze angelangt. Nach unten blickend können wir die gebräuchlichsten Sprachen beurteilen.

(Matthias Rosin/ev)



Forth – die etwas andere Programmiersprache

Wenn man Forth lernen möchte, braucht man dazu zweierlei: ein Forth-System und ein Buch zum Lernen. Der folgende Artikel soll Ihnen den Einstieg in Forth lediglich erleichtern; wir sagen Ihnen, welche Literatur geeignet ist und worauf Sie bei einem Forth-Compiler für Ihr Computersystem achten müssen.

Sicher haben Sie schon den einen oder anderen Artikel über Forth gelesen. Dort war häufig die Rede von der »umgekehrt polnischen Notation«, vom Stackkonzept und von Worten wie SWAP, DUP und ROT, die dem Uneingeweihten allenfalls ein Stirnrunzeln entlocken. Über das Wesen der Sprache sagen sie aber nichts aus. Warum sich Forth in letzter Zeit dennoch zum »Geheimtip« unter Programmierern entwickelte, hat andere Gründe. Forth ist sicher nicht die »eierlegende Wollmilchsau«, wie dies von manch anderen Programmiersprachen behauptet wird, bietet aber einige Vorteile, die eine genauere Betrachtung rechtfertigen. Eine Warnung vorweg: Forth ist in vieler Hinsicht ungewöhnlich und sicher nicht jedermanns Sache. Wenn Sie Angst vor den Innereien Ihres Computers oder Systemabstürzen haben, dann ist Forth nichts für Sie. Arbeiten Sie jedoch gern maschinen-

nah, um die letzten Feinheiten aus Ihrem Computer herauszuholen, und können Sie mit Bits und Bytes etwas umgehen, dann lesen Sie weiter.

Allen höheren Programmiersprachen ist eins gemeinsam. Damit der Prozessor, das Herzstück eines jeden Computers, versteht, was der Programmierer will, muß der Programmtext übersetzt werden. Letztlich kann der Computer immer nur seinen Maschinencode verstehen.

Programmiersprachen lassen sich in zwei Kategorien einteilen. Da sind auf der einen Seite die Interpretersprachen wie Basic, Logo und Comal. Bei diesen Sprachen wird ein Programm während der Ausführung übersetzt (interpretiert). Dieses Verfahren hat den Vorteil der Interaktion, das heißt, man kann die Wirkung eines Befehls oder einer Befehlsfolge sofort sehen. Programme entwickeln geht also verhältnismäßig schnell, und auch die unvermeidliche Fehlersuche ist kein Problem. Man läßt einfach sein Programm Schritt für Schritt ablaufen, um zu sehen, wann und wo der Fehler auftritt. Nachteil dieser Methode ist jedoch die geringe Ablaufgeschwindigkeit der Programme. Schließlich beschäftigt sich der Computer den größten Teil seiner Rechenzeit mit der Übersetzung und nicht mit der Bearbeitung des eigentlichen Programms. Jeder, der einmal ein längeres Basic-Programm mit einem entsprechenden Programm in Maschinencode verglichen hat, wird dies bestätigen.

Aus diesem Grund schlägt man bei den Compilersprachen wie Pascal, C oder Modula einen anderen Weg ein. Der Programmtext wird ein einziges Mal in Maschinencode übersetzt (compiliert) und kann dann immer wieder ausgeführt werden. Das klingt gut, hat aber natürlich ebenfalls seine Tücken. So ist es ein langer Weg vom Quelltext bis zum lauffähigen Programm. Typisch für solche Sprachen ist der Dreischritt Editor, Compiler, Linker, der sich immer wiederholt und viel Zeit kostet. Ein Programmfehler wird ja fast immer erst ganz am Schluß erkannt, wenn der Computer sang- und klanglos abstürzt.

Forth verbindet die Vorteile von Interpreter- und Compilersprachen, also hohe Ablaufgeschwindigkeit und interaktive Programmentwicklung. In Forth können Sie die Wirkung jedes Befehls wie in Basic unmittelbar überprüfen, die Programme laufen jedoch zirka zehnmal schneller ab.

Zudem steht es offen, zeitkritische Programmteile mit dem fast immer vorhandenen Assembler unmittelbar in Maschinencode zu schreiben. Solche Routinen sind in der Regel sehr kurz. Der Forth-Interpreter/Compiler behandelt sie aber genauso wie alle anderen Forth-Befehle auch; Sie stoßen also auf keine SYS- oder CALL-Sequenzen mit irgendwelchen unverständlichen Zahlen dahinter.

Forth ist vollkommen strukturiert. Ein Programm besteht aus einer Reihe von Befehlen, in Forth Wörter genannt, die jeweils einzeln entwickelt und getestet werden. Jeder so neu erzeugte Befehl wird durch seine Definition dem Sprachkern hinzugefügt und steht damit zur Bildung weiterer Wörter bereit. Für Verknüpfungen stehen die von Pascal bekannten Kontrollstrukturen wie IF..ELSE..THEN, BEGIN..WHILE ..REPEAT, DO..LOOP und so weiter zur Verfügung. GOTO und GOSUB fehlen ebenso wie Zeilennummern. Die Wörter werden einfach durch Nennung ihres – hoffentlich sinnvollen – Namens aufgerufen. Das Ergebnis ist ein übersichtlicher, wartungsfreundlicher Code. Das lernt man spätestens dann zu schätzen, wenn man sich nach einigen Wochen oder Monaten ein Programm zur Überarbeitung wieder vornimmt. Haben Sie das einmal mit einem schlecht dokumentierten Assemblerprogramm versucht, wissen Sie das zu schätzen.

In der Kürze liegt die Würze

Forth macht nahezu alles möglich, insbesondere den Zugriff auf die gesamte Hardware. Viele Sprachen schieben da einen Riegel vor, entscheiden für den Programmierer, was erlaubt ist und was nicht. Forth ist hier genauso wie Assembler und bietet übrigens auch die gleichen Fehlerquellen. So sind Ein-Ausgabe-Bausteine unter Forth ebenso leicht zu programmieren, wie man sich eigene Speicherverwaltungen abseits von gewöhnlichen Variablen oder Arrays zusammenbauen kann. Ja, sogar der Forth-Compiler ist offen für Veränderungen. Dieser uneingeschränkte Zugang verlangt natürlich viel Disziplin beim Programmieren. Allerdings: Mehr als abstürzen kann Ihr Computer auch unter Forth nicht.

Forth-Programme sind kurz, in der Regel sogar kürzer als entsprechende Assembler-Programme. Das liegt an der Arbeitsweise des Forth-Compilers. Im Speicher steht bei jedem Wort nur eine Liste mit Adressen der Worte, aus denen es sich zusammensetzt. Der innerste Kern des Forth-Interpreters – übrigens eine Maschinencodesequenz von zirka 10 Byte – sorgt für die Abarbeitung dieser Liste. Das ist die Grundidee. Natürlich gibt es eine Reihe von Feinheiten. So kommt es, daß »normale« Forth-Systeme mit 300 bis 400 Befehlen Grundwortschatz nur zirka 10 bis 20 KByte Speicher benötigen.

Nachdem wir Ihnen nun den Mund hoffentlich ausreichend wäßrig gemacht haben, wenden wir uns wieder der eingangs gestellten Frage zu. Neben einem Computer braucht man, so hieß es dort, ein Forth-System und Literatur. Die Frage nach der Literatur läßt sich verhältnismäßig leicht beantworten. Es gibt ein für Anfänger wie Fortgeschrittene gleichermaßen geeignetes Buch. Es heißt »Starting Forth« von Leo Brodie. Eine deutsche Übersetzung ist unter dem Namen »Programmieren in Forth« im Hanser-Verlag erschienen. Das Buch entwickelte sich zu so etwas wie einem Standardwerk, weil es locker und doch exakt geschrieben ist. Wer gut Englisch kann, dem sei auf jeden Fall die Originalausgabe empfohlen, weil sich in der deutschen Übersetzung, vor allem bei den Programmbeispielen, einige Fehler eingeschlichen haben. Man sollte es allerdings nicht abends im Bett kurz vor dem Einschlafen lesen, sondern neben den Computer legen und damit arbeiten. Auch Forth lernt man, indem man in Forth programmiert und nicht beim Lesen.

Womit wir bei der Frage eines Forth-Systems wären. Hier läßt sich natürlich keine allgemeingültige Lösung angeben, da es für jeden Computer verschiedene Versionen gibt. Vor einiger Zeit veröffentlichte die Zeitschrift »Forth-Dimensions« (von der amerikanischen »Forth Interest Group« herausgegeben), eine Art Checkliste für Forth-Systeme. Wir bringen zu Ihrer Orientierung eine deutsche Übersetzung, damit Sie wissen, worauf man achten sollte. Maximal sind dreizehn Punkte zu vergeben. Systeme mit weniger als sieben Punkten dürften Ihnen die Arbeit mehr erschweren als erleichtern, von ihnen sollten Sie lieber die Finger lassen.

Die Größe eines Systems entscheidet über den Umfang, also die Anzahl der Befehle, die im Grundwortschatz enthalten sind. Sicher ist die reine Länge in KByte kein ausreichendes Merkmal, wichtiger ist die Befehlsanzahl. Sie gibt aber doch einigen Aufschluß, ob es sich um eine reine Standardimplementation handelt, die mit zirka 10 KByte auskommt, oder ob etwas Komfort geboten wird. Auch läßt sich aus der Länge in etwa beurteilen, wie viele der Grundworte in Maschinencode geschrieben sind. Zwar etwas länger, wirken sie sich aber positiv auf die Laufzeit des Systems aus. Trotzdem: Seien Sie vorsichtig, wenn Ihnen Systeme mit 30 KByte und mehr Länge angeboten werden. Diese enthalten normalerweise viel überflüssigen Ballast.

Viel wichtiger ist dagegen der nächste Punkt. Viele Systeme enthalten



Zusätze in Form von Quelltexten. Das hat zwei Vorteile. Zum einen muß man nur die wirklich benötigten Teile laden, zum anderen kann man die Quelltexte seinen Wünschen anpassen und verändern. Nebenbei bemerkt lernt man bekanntlich aus Beispielen am besten: Wem solche Beispiele gleich mitgeliefert werden, der hat es einfacher. Zusätze enthalten Programmierhilfen wie Decompiler, Einzelschrittracer, Assembler und so weiter oder auch fertige Programme.

Wie steigt man ein?

Die nächsten beiden Punkte beziehen sich auf den Editor. Wir halten dies für eins der wichtigsten Hilfsmittel jeder Programmiersprache. Schließlich arbeiten Sie beim Programmieren fast ausschließlich mit dem Editor. Sein Komfort entscheidet über die Bedienbarkeit eines Systems. Entspricht der Editor dem in Starting Forth beschriebenen, hat das den Vorteil, daß Sie die Beispiele aus dem Buch leichter bearbeiten können. Ein guter Editor sollte bildschirmorientiert arbeiten, das heißt, man kann mit dem Cursor auf dem Bildschirm »umherwandern« und Änderungen sofort sehen. Der Starting-Forth-Editor verfügt über diese Eigenschaften von Haus aus nicht, es gibt aber entsprechend erweiterte Versionen. Ein zeilenorientierter Editor ist nur zu empfehlen, wenn man noch nie mit etwas anderem gearbeitet hat.

Zu den nächsten beiden Punkten: Alljährlich treffen sich Forth-Programmierer, die diese Sprache professionell nutzen, in Amerika, um über Veränderungen oder Erweiterungen des Forth-Sprachkerns zu beraten. Dabei werden die Erfahrungen, die sich aus der praktischen Arbeit mit Forth ergaben, aufgearbeitet und gleichzeitig neue Ansätze diskutiert. Wenn Einigkeit über eine notwendige Änderung herrscht, wird ein neuer Standard festgelegt. Der letzte Standard wurde 1983 beschlossen. Davor gab es einen im Jahr 1979. Zwar erscheinen die Änderungen – vor allem dem Anfänger – häufig spitzfindig. Sie sorgen jedoch für eine stetige Weiterentwicklung der Sprache und verhindern das Auseinanderfallen in verschiedene Dialekte, wie es zum Beispiel bei Basic geschehen ist. Ein Programm, das nur Standard-Definitionen enthält, hat den unschätzbaren Vorteil, daß es unabhängig vom verwendeten Computer läuft. So lassen sich Programme für die verschiedensten Computer untereinander austauschen, jeder kann von Erfahrungen anderer profitieren. Die folgenden Punkte beziehen sich auf die Beschreibung und Unterstützung Ihres Systems. Eine Beschreibung ist unerläßlich, denn jedes Forth-System verfügt über eine Reihe von systemspezifischen Erweiterungen. Der Standard legt nur zirka 150 Worte fest, ein übliches Forth-System enthält aber 300 bis 400 Worte. Darüber hinaus sind häufig Besonderheiten und Zusätze eingebaut, die man natürlich nur mit einer vernünftigen Beschreibung ausnutzen kann. Diese sollte auch Informationen darüber enthalten, wie der Compiler arbeitet, welchen Speicher das System benutzt und so weiter. Je mehr Sie über Ihr Forth-System wissen, desto besser, auch wenn Sie mit den Informationen zunächst nicht viel anfangen können. Von einem guten Händler können Sie auch erwarten, daß er auf Ihre Fragen und Probleme eingeht.

Die letzten Punkte der FIG-Checkliste behandeln spezielle Erweiterungen. Ein Assembler sollte zur Grundausstattung jedes Systems gehören; ein Fließkommapaket wird man dagegen nur selten brauchen. Forth arbeitet aus Geschwindigkeitsgründen fast ausschließlich mit Integerarithmetik. Sogar trigonometrische Funktionen, wie man sie für grafische Anwendungen häufig braucht, lassen sich ohne Fließkomma programmieren. Zugriff auf ein File-System sollte möglich sein, weil es sonst schwierig wird, von Forth aus auf Files einer Textverarbeitung oder einer Tabellenkalkulation zuzugreifen. Forth selbst arbeitet normalerweise direkt auf Diskette, mit physikalischem Zugriff ohne File-System.

Wo bekommt man ein Forth-System?

Die verschiedensten Firmen bieten Forth an, zum Teil zu horrenden Preisen. Dabei muß der Preis kein Qualitätsmerkmal sein, im Gegenteil: Einige der besten Forth-Compiler sahen ihre Autoren als »public domain« (also der Allgemeinheit kostenlos zugänglich), was manche Vertreiber nicht davon abhält, sich die Anpassung auf ein spezielles Computersystem teuer bezahlen zu lassen. Den geringen Umsatz versucht man dann über hohe Preise auszugleichen.

Es gibt in Deutschland einen »Ableger« der »Forth Interest Group«, die »Forth Gesellschaft e.V.« in Hamburg. Sie setzt sich als Aufgabe die Verbreitung der Programmiersprache Forth. Die Gruppe arbeitet nicht kommerziell, sondern finanziert sich aus Beiträgen und Spenden. Dort kann man sich über Forth-Systeme für die verschiedenen Computer informieren, auch werden Bezugsquellen genannt. Es existiert auch eine Sammlung der verschiedensten Artikel über Forth, die man sich gegen Unkostenerstattung kopieren lassen kann. Für Mitglieder erscheint eine Zeitung namens »Vierte Dimension«, die zweimonatlich erscheint. Wer sich an die Forth-Gesellschaft wenden möchte, kann dies unter folgender Adresse tun:

Forth-Gesellschaft e.V.
Schanzenstr. 27
2000 Hamburg 6

Zum Abschluß noch ein Bonbon: Für den Commodore 64 und den Atari ST gibt es das »volksFORTH-83«, ein Forth-System, das von Mitgliedern der Forth-Gesellschaft geschrieben wurde und ebenfalls »public domain« ist. Es handelt sich um eins der besten Forth-Systeme, die es gegenwärtig gibt: Es entspricht vollständig dem 83'er-Standard, enthält Fullscreen-Editor, Assembler und eine Fülle von Tools, angefangen vom Decompiler bis hin zu einem Grafik-Paket. Das System ist multi-tasking-fähig, das heißt mehrere Programme können gleichzeitig ablaufen. Der Quelltext ist einschließlich des System-Quelltextes verfügbar und das System frei kopierbar. Die Weitergabe ist sogar ausdrücklich erwünscht. Wen es interessiert, kann es auch bei der Forth-Gesellschaft zu einem Selbstkostenpreis von 45 bis 65 Mark kaufen; man erhält dann – je nach Computersystem – mehrere Disketten mit sämtlichen Quelltexten sowie ein 200-seitiges Handbuch. Derzeit sind Versionen für Z80-, 8080- und 8068-Prozessoren in Arbeit. Nach der FIG-Checkliste erhält »volksForth-83« 12 von 13 möglichen Punkten.

Ebenfalls erwähnen wollen wir das F83 von Henry Laxen und Michael Perry. Auch hierbei handelt es sich um ein »public domain«-System nach dem 83'er-Standard. Es sind auch sämtliche Quelltexte zum System und allen Erweiterungen zu erhalten. F83 gibt es für 8080-, 8086- und 68000-Prozessoren. Bei der Forth-Gesellschaft erhalten Sie auf jeden Fall eine Version für den IBM-PC und kompatible Rechner. Zum F83 liegt keine Dokumentation vor. Statt dessen enthält die Version für jeden Quelltext Kommentarscreens. Auch dieses System erhält laut Checkliste 12 Punkte.

Es steht nun nichts mehr im Wege, eigene Erfahrungen zu sammeln. Beschaffen Sie sich Starting Forth und ein Forth-System, und setzen Sie sich eine Woche oder sagen wir bis zum dreißigsten Systemabsturz an Ihren Computer. Entweder werfen Sie dann, dem Wahnsinn nahe, alles in die Ecke, oder aber die Faszination dieser Sprache hat Sie gepackt und läßt Sie so schnell nicht mehr los.

(Dietrich Weineck/hg)

Forth: Programmieren in der vierten Dimension

Eine Programmiersprache, die sich in der letzten Zeit ständig steigender Beliebheit erfreut, ist Forth. Zu Recht, denn Forth ist eine äußerst leistungsfähige Sprache mit einem ungewöhnlichen Konzept. Schwer zu lernen ist sie nicht.

Seit es Computer gibt, ist das zentrale Problem die Kommunikationsschnittstelle Mensch/ Computer – oder anders formuliert, wie sage ich meinem Computer, was er tun soll? Während in der »grauen Vorgeschichte« der EDV noch bitweise programmiert werden mußte, erwies sich dieses Verfahren im Laufe der Zeit als viel zu umständlich. Es entstanden höhere Programmiersprachen, die sich einer bestimmten Anzahl von Befehlsworten bedienen. Meist sind diese an die menschliche Sprache angelehnt. Zu solchen höheren Programmiersprachen zählen beispielsweise Fortran, Basic, Pascal, C oder Lisp, um nur einige zu nennen.

Betrachtet man sich die Entwicklung in der Computertechnik, so stellt man fest, daß sich, seitdem das erste Röhrengerät in Betrieb ging, bis zum heutigen Tage die Hardware rasant weiterentwickelt hat. Anders die Software. Hier dominieren noch immer Sprachen wie Fortran und Basic, die in ihren Ursprüngen in die fünfziger Jahre zurückreichen.

Daß Programmiersprachen für Computer, die Sie heute nur noch im Museum bewundern können, nicht immer den heutigen Ansprüchen genügen, ist leicht einzusehen. Vor allem Basic, das sich mittlerweile zum Standard für kleine und mittlere Systeme entwickelt hat, hinkt den Ansprüchen der meisten Programmentwickler weit hinterher. Zwar ist es leicht zu erlernen und besitzt eine unkomplizierte Befehlssyntax, aber strukturierte Programme, lokale Variablen und ähnliches sind für die meisten Dialekte tabu.

Was tut also der (Basic-)frustrierte Programmierer? Er sucht sich eine neue Sprache, die seinen Ansprüchen besser gerecht wird. Und da beginnt das Dilemma. Welche der vielen Konkurrenten kommt für ihn in Frage? Hier ist es hilfreich, sich zu überlegen, was die »neue Traumsprache« alles leisten soll. Folgende Punkte sind dabei für jeden Programmierer wichtig. Die Sprache soll:

- schnell sein,
- strukturiertes Programmieren ermöglichen,
- auf andere Computertypen übertragbar sein,
- die Stärken des eigenen Geräts unterstützen,
- erweiterbar sein,
- möglichst wenig des kostbaren RAM-Speichers belegen,
- relativ leicht zu erlernen und zu verstehen sein.

Wenn das auch Ihre Vorstellungen einer nahezu idealen Programmiersprache sind, dann brauchen Sie nicht länger zu suchen. Denn solch eine Traumsprache gibt es schon – Forth.

Zwar ist auch Forth nicht die Programmiersprache schlechthin, doch weist sie zumindest die oben aufgeführten Vorteile (und noch einige mehr) auf. Was es damit tatsächlich auf sich hat, darüber klären Sie die nächsten Seiten auf. Sie ersetzen zwar kein Handbuch von Forth (es können auf so wenig Seiten niemals alle Anweisungen erklärt werden), aber wir wollen versuchen, Ihnen ein Gefühl für den typischen Charakter von Forth zu geben. Vielleicht kommen Sie auf den Geschmack, sich mit dieser faszinierenden Sprache näher zu befassen.

Forth, Sprache für den Weltraum

Forth wurde Ende der sechziger Jahre entwickelt und ursprünglich zur Steuerung und Auswertung der Meßdaten einer Sternwarte eingesetzt. Schon damals stand die Zukunft von Forth im wahrsten Sinne des Wortes in den Sternen, denn die amerikanische Weltraumbehörde NASA wählte Forth als Sprache für zukünftige Satellitenprogramme. Doch bevor es dazu kommen sollte, blieb es bis Ende der siebziger Jahre sehr ruhig um diese Sprache. 1977 gründete eine nichtkommerzielle Gruppe von Programmierern die »Forth Interest Group« (FIG) und machte Forth damit einer breiteren Öffentlichkeit zugänglich. Es entstand als Standard FIG-Forth, aus dem sich zwei Jahre später der Forth 79-Standard entwickelte. Beide Versionen sind heute im Heim- und Personal Computerbereich verbreitet. Der Unterschied dieser Dialekte ist nur gering, so daß sich FIG-Forth-Programme leicht in den 79-Standard umsetzen lassen – und umgekehrt.

In unserer Einführung wollen wir im wesentlichen das ältere FIG-Forth benutzen. Etwaige Abweichungen zum 79-Standard bleiben aber auch nicht unerwähnt.

Mittlerweile wird für fast jedes Computersystem eine Forth-Version angeboten. Da die Sprache kaum Speicherplatz beansprucht, ist sie ideal für kleinere Systeme geeignet. Im allgemeinen benötigt Forth nicht mehr als 10 KByte RAM-Bereich. Je nachdem, welche Extras zusätzlich implementiert sind, kann sich dieser Bereich natürlich erhöhen.

Forth kommt im Prinzip ohne Massenspeicher aus, weshalb sich ein Diskettenlaufwerk erübrigt. Da beim Heimcomputer die Programme nicht im ROM vorliegen, braucht man aber unbedingt einen Kassettenrecorder. Dennoch, wie bei allen Sprachen ist auch unter Forth der Gebrauch eines Diskettenlaufwerks angenehmer als ein Datenrecorder. Auch kommen einige Stärken von Forth nur mit einem Laufwerk zum Tragen.

Nachdem Sie Ihre Forth-Version geladen und gestartet haben, erscheint, zusammen mit einer Mitteilung über die Herkunft des Systems, der gleiche Cursor, den Sie sicher schon von Basic her bestens kennen.

Zwei verschiedene Wege gibt es, um sich mit der neuen Sprache vertraut zu machen. Entweder Sie starten einen »Trockenkurs« und studieren das Handbuch in allen Einzelheiten, oder aber Sie legen alle Hemmungen ab, geben irgendetwas ein und warten, was der Computer machen wird. Entscheiden Sie sich für die zweite Art, so werden Sie sehr oft eine nüchterne Fehlermeldung auf dem Bildschirm vorfinden. Diese besteht entweder aus einem Kommentar wie zum Beispiel »CAN'T FIND« oder aus einer Ziffer, die die Fehlerart anzeigt. Manchmal wird der Computer aber auch mit einem lapidaren »OK« antworten. Immer dann haben Sie eine Forth-Anweisung richtig benutzt.

Für den Fall, daß Sie aber den Wort-



schatz ohne langes Ausprobieren kennenlernen wollen, ist in fast jedem Forth-Compiler eine Anweisung vorgesehen, die alle Befehle auf dem Bildschirm ausgibt. »VLIST« oder »WORDS« sind zwei häufig gebrauchte Kennworte für diesen Befehl. Dieses Wörterbuch zeigt sämtliche Anweisungen, die Ihr Compiler versteht. Ganz egal, ob es sich um vor- oder selbstdefinierte Befehle handelt.

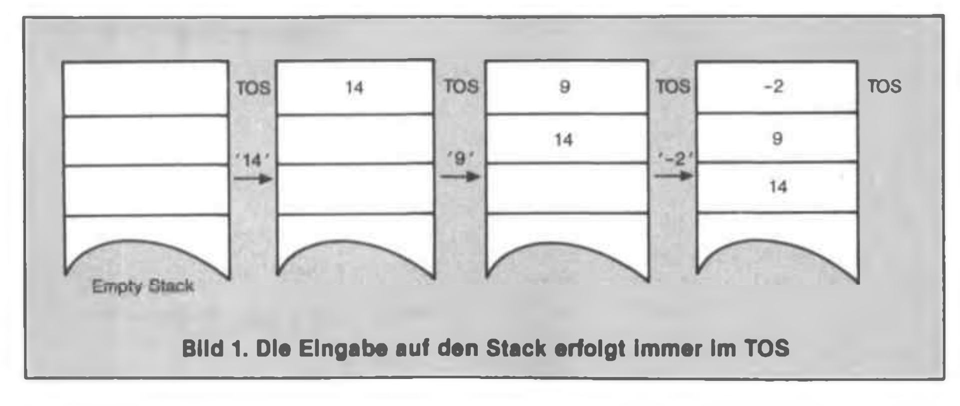


Bild 1. Die Eingabe auf den Stack erfolgt immer im TOS

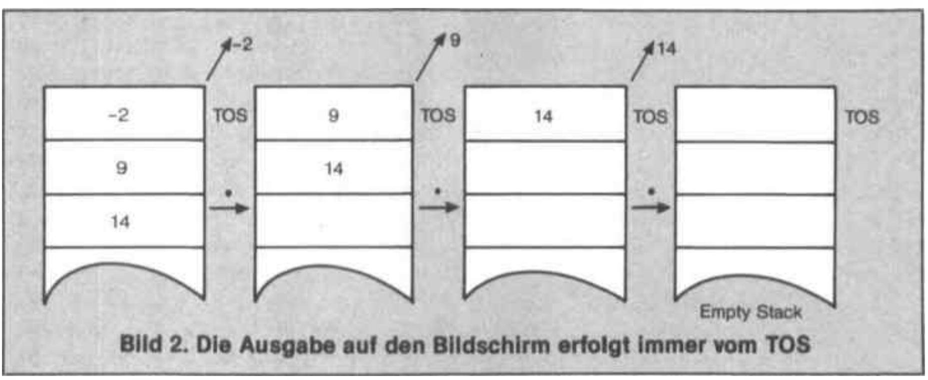


Bild 2. Die Ausgabe auf den Bildschirm erfolgt immer vom TOS

Erste Kontaktaufnahme

Unter Forth müssen alle Eingaben mit der RETURN- (oder ENTER-) Taste abgeschlossen werden. Die eingegebenen Worte lassen sich in drei Gruppen unterscheiden – in »Nonsens«-Worte, die der Compiler mit einer Fehlermeldung quittiert, in Anweisungen oder in Zahlen. Denn auch wenn Sie nur eine Zahl eingeben, reagiert der Compiler darauf mit »OK«, das heißt, er akzeptiert den Befehl ohne Probleme. Was ist nun mit dieser Zahl geschehen?

Bei nahezu allen Operationen in Forth spielt der Stack eine zentrale Rolle. Der Stack (zu deutsch Stapelspeicher) ist nichts anders als ein kleiner Speicherbereich, der nach einem besonderen Prinzip verwaltet wird. Jede Zahleneingabe von der Tastatur, die mit RETURN abschließt, landet zuerst einmal im »Top Of Stack« (TOS), also in der obersten Speicherzelle (eine Speicherzelle ist ein 16-Bit-Register) des Stacks.

Jede neu hinzukommende Zahleneingabe wird ebenfalls dort abgelegt. Alle bereits vorhandenen Zahlen rutschen um eine Position nach unten. Bei der Ausgabe vom Stack kommt als erstes die Zahl im TOS an die Reihe. Da es sich hierbei immer um die zuletzt eingegebene Zahl handelt, wird das Ganze als »Zuletzt rein – zuerst raus«-Prinzip bezeichnet. Im Fachenglisch heißt das dann »Last In – First Out«-Prinzip (kurz LIFO).

Alle Operationen, die den Stack beeinflussen, arbeiten nach diesem Verfahren. Deswegen sollten Sie sich damit gut vertraut machen. Bereitet Ihnen die Arbeitsweise noch Schwierigkeiten, dann hilft Ihnen vielleicht folgender Vergleich weiter.

Stellen Sie sich einen Schreibtisch vor, auf dem ein Stoß Papier liegt. Legen Sie ein Blatt auf dem Stoß ab, so landet es auf der obersten Stelle. Das nächste Blatt liegt darüber und an oberster Position befindet sich damit immer das Blatt, das zuletzt abgelegt wurde. Wollen Sie nun den Papierstoß abarbeiten, dann nehmen Sie zuerst das oberste Blatt weg. Und dieses ist das zuletzt hingelegte. Das schon am längsten auf dem Tisch liegende Papier ist das letzte beim Abarbeiten. Und so funktioniert auch der Stack in Ihrem Computer.

Nach soviel Theorie zurück zu Forth. Wir wissen nun, wie der Stack verwaltet wird und wollen ausprobieren, wie wir drei Zahlen auf den Stack legen und wieder herunterholen können. Wir geben einfach folgende Zeile ein:

```
14 9 -2
```

Zwischen zwei Zahlen muß immer ein Leerzeichen stehen und die Zeile mit der RETURN-Taste beendet werden. Der Computer quittiert die Eingabe mit »OK«. Was ist nun aber auf dem Stack passiert? Dazu betrachten wir Bild 1, das uns zuerst den »leeren« Stack und zum Schluß den »vollen« Stack zeigt. Zuerst wurde die 14 im TOS abgelegt, dann die 9 (gleichzeitig wandert die 14 um eins nach unten) und zum Schluß die -2 (die andern beiden Werte wandern nach unten).

Um diese Zahlen wieder vom Stack auszugeben, lernen Sie nun Ihr erstes Forth-Wort kennen (Befehle, bezie-

Forth-Steckbrief

- Forth wurde Ende der sechziger Jahre in den USA entwickelt.
- Forth ist eine Compilersprache, mit der Sie aber auch interaktiv arbeiten können.
- Forth ist sehr schnell. Vergleichbare Programme brauchen in Basic bis zu 20mal mehr Zeit als in Forth.
- Der Befehlssatz in Forth besteht aus 200 bis 300 Wörtern. Ein Wort läßt sich mit einem Unterprogramm in Basic oder besser mit einer Prozedur in Pascal vergleichen. Der Benutzer kann diesen Wortschatz um eigene Definitionen erweitern. Diese neuen Wörter sind im Gebrauch mit den Standard-Forth-Wörtern vollkommen identisch.
- Bei Forth-Wörtern handelt es sich entweder um Secondaries, die wiederum aus Forth-Wörtern aufgebaut sind, oder um Primitives, welche in Maschinencode definiert sind.
- Forth rechnet nur mit Integerzahlen von 16 oder 32 Bit Breite.
- Forth belegt in der Regel zwischen 8 und 12 KByte Speicherplatz (hängt von dem Umfang des Wortschatzes ab).
- Forth verwaltet den Diskettenspeicher virtuell, das heißt RAM- und Diskettenspeicher sind formell gleichwertig.
- Forth ist weitgehend standardisiert, das heißt alle für Heim- und Personal Computer angebotenen Versionen leiten sich vom FIG-Forth ab. FIG-Forth stammt von der Forth Interest Group, einer nichtkommerziellen Vereinigung von Programmierer.
- Die Rechenoperationen werden in Forth nach den Regeln der Umgekehrten Polnischen Notation (UPN) durchgeführt.

hungsweise Anweisungen, werden in Forth als Wort bezeichnet). Es handelt sich um einen unscheinbaren Punkt. Durch ».« wird die Zahl im TOS (Top Of Stack) auf den Bildschirm ausgegeben.

```
. -2 ok
```

Nach der Eingabe des Punktes und der RETURN-Taste reagiert der Computer mit der Meldung »-2 OK«. Wir wollen im folgenden immer die Zeile so angeben, wie Sie nach der Bearbeitung aussieht. Eingeben dürfen Sie natürlich nur die Forth-Wörter (in diesem Falle also nur den Punkt). Sie können natürlich auch mehrere Zahlen auf einmal ausgeben lassen:

```
. . 9 14 OK
```

Der nächste Punkt veranlaßt den Computer zu einer Fehlermeldung, da der Stack leer ist:

```
. 0 EMPTY STACK
```

Bild 2 zeigt Ihnen, wie sich der Stack bei der Ausgabe der Zahlen verändert. Sie sehen dabei, daß jede Zahl, die durch ».« auf dem Bildschirm erscheint, gleichzeitig vom Stack verschwindet und sich alle anderen Zahlen um eine Position nach oben bewegen.

Bevor wir zu unseren ersten Rechenaufgaben in Forth kommen, müssen wir uns noch einmal ganz genau mit der Syntax von Forth auseinandersetzen. Befehlswörter dürfen in Forth beliebig in einer Zeile stehen. Allerdings muß immer mindestens ein Leerzeichen zwei Anweisungen trennen. Anders als beispielsweise in Basic, wo es einen festgelegten, im Grunde nicht mehr erweiterbaren Befehlssatz gibt, kann jedermann Forth um neue Befehle bereichern. Da dabei jede Zeichenkombination als Wortname erlaubt ist, stellen die Leerzeichen für den Textinterpreter die einzige Möglichkeit dar, die Wörter voneinander zu unterscheiden.

. (n –)	–gibt die Zahl im TOS (oberste Zahl im Stack) aus
. »Text«	–gibt die Symbole zwischen den Anführungszeichen als Text aus
CR	–bewirkt einen Zeilenvorschub

Tabelle. Ihre ersten Wörter in Forth

Auch Texte lassen sich unter Forth auf den Bildschirm ausgeben. Dazu dient ein zweites Wort: »."«. Ein Beispiel:

```
." OSTERHASE" OSTERHASE OK
```

Wenn Sie ».''OSTERHASE''« mit RETURN an den Computer abschikken, dann gibt er den Text »OSTERHASE« zurück. Denken Sie an die richtige Verteilung der Leerzeichen, da sonst der Computer Sie nicht verstehen kann.

Mit solch einer Textausgabe können wir auch unsere Stack-Ausgabe komfortabler gestalten:

```
4 OK
CR ." TOS : " .
TOS : 4 OK
```

Wenn Sie die 4 und die zweite Zeile eingegeben haben, dann antwortet der Computer mit der dritten. Das neue Wort »CR« bewirkt einen Zeilenvorschub. Bei manchen Compilern funktioniert dieser letzte Befehl nur in Wörtern. Wie Sie solche definieren, erfahren Sie später. Manchmal darf ».''...''« durch ».(...)« ersetzt werden. Näheres finden Sie in Ihrem Handbuch. Die Wörter des ersten Teils der Einführung in Forth faßt die Tabelle noch einmal zusammen.

(Peter Monadjemi/hg)

UPN – Rechnen in der umgekehrten Polnischen Notation

Forth zeigt einige unkonventionelle Lösungswege, Computerprogramme zu erzeugen. Besonders das Rechnen in Forth unterscheidet sich von fast allen anderen Computersprachen.

Der Stack ist das wichtigste Hilfsmittel zum Rechnen in Forth. Dazu stehen verschiedene Operatoren und Befehlswörter zur Verfügung. Allerdings muß man bei dem Jonglieren mit Zahlen in Forth einige spezifische Besonderheiten beachten:

- Alle Operationen werden nach den Regeln der Umgekehrten Polnischen Notation (UPN) durchgeführt.
- Forth rechnet nur mit Integerzahlen (ganze Zahlen).
- Forth kennt in der Standard-Version nur die vier Grundrechenarten.

UPN ist eine Vorschrift für die Durchführung von Rechenoperationen unter Einbeziehung eines oder mehrerer Stacks. Nehmen Sie als Beispiel die Operation »33*4=« (hier noch in der Basic-typischen Schreibweise). Die UPN-Schreibweise bereitet die Befehlszeile so auf, wie sie der Computer am leichtesten bearbeiten kann. Das bedeutet, daß zuerst die Operanden (Zahlen) und dann die Operatoren (Rechenzeichen) eingegeben werden. Unser Beispiel sieht dann wie folgt aus:

```
33 4 *
```

Das Ergebnis befindet sich nach der Berechnung im Stack – und zwar im TOS (Top of Stack) – und kann dort weiterverarbeitet werden. Der Vorteil der UPN gegenüber der Infix-Notation (das ist die Basic-übliche Eingabe von Berechnungen) macht sich erst bei größeren Ausdrücken bemerkbar. Die Eingabe in Infix-Schreibweise:

```
(2+7)/(4*(8-3))=
```

braucht bedeutend mehr Platz als die UPN-Schreibweise:

```
2 7 + 4 8 3 - * /
```

Sie sehen, daß die UPN weder Klammern noch Gleichheitszeichen benötigt. Dadurch ergibt sich neben einer kürzeren, Speicherplatz sparenden Schreibweise auch ein erheblicher Geschwindigkeitsvorteil. Die Klammern erfordern vom Computer nämlich immer ein Vorausschauen, »wo wird diese wieder geschlossen«. Bei der UPN hingegen werden bei jedem Rechenoperator die beiden obersten Werte im Stack miteinander verknüpft und deren Ergebnis direkt im TOS abgelegt. Für unsere Aufgabe bedeutet das, daß mit Eingabe des Pluszeichens die 2 und 7 addiert und das Ergebnis 9 in den TOS gelegt wird. Die Eingabe der drei Zahlen 4, 8 und 3 bewirkt, daß die 9 an der vierten Stelle von oben liegt. Das Minuszeichen berechnet 8 minus 3, legt die 5 in den TOS und zieht die beiden anderen

Werte nach oben. Das Malzeichen multipliziert die 5 mit der 4, legt das Ergebnis ab und schon steht die 9 direkt unter der 20. Das Divisionszeichen besorgt den Rest, so daß zum Schluß das Ergebnis der Rechnung im TOS steht. Die UPN ist also nicht etwa ein exotisches Rechenverfahren, sondern die »natürlichste« und effektivste Methode für einen Computer, Rechenoperationen zu verarbeiten.

Nun wollen wir uns aber damit befassen, wie sich der Stack verändert, wenn wir die Grundrechenarten bearbeiten lassen. Die Addition löst das Zeichen »+« aus.

```
4 5 + OK
```

Was ist nun auf dem Stack geschehen? Dazu betrachten wir uns Bild 1. Sie sehen, daß sich vor dem Aufruf von »+« die beiden zu addierenden Zahlen an den beiden obersten Stellen des Stacks befinden müssen. Nach der Addition wird das Ergebnis im TOS abgelegt. Von dort kann man es mit ».« leicht auf den Bildschirm ausgeben.

```
. 9 OK
```

Die 9 zeigt an, daß das Ergebnis tatsächlich im TOS gespeichert war. Die Subtraktion erfolgt analog:

```
16 12 - . 4 OK
```

Auch hier müssen sich die beiden Zahlen zuerst auf dem Stack befinden. »-« zieht die Zahl im TOS von der darunterliegenden ab (siehe Bild 2) und »/« ruft die Division auf (Bild 3):

```
20 2 / . 10 OK
```

Bis dahin ist die Welt noch in Ordnung. Doch das nächste Beispiel führt zu einem unerwarteten Ergebnis:

```
21 2 / . 10 OK
```

Hier kommt das schon erwähnte Fehlen von Real-(Fließkomma-)zahlen voll zum Tragen. Aber keine Bange: Forth bietet verschiedene Wege, auch beliebig genaue Ergebnisse zu erhalten. Mit dem Operator »/« bekommen Sie also nur die Vorkommazahl. Um den ganzzahligen Rest dieser Rechenoperation zu erhalten, gibt es unter Forth einen Befehl, den die meisten Basic-Dialekte nicht kennen.

```
21 2 MOD . 1 OK
23 4 MOD . 3 OK
```

21 geteilt durch 2 gibt 10, Rest 1 und 23 geteilt durch 4 gibt 5, Rest 3. Die ganzzahligen Restbeträge werden bei diesem Wort in den TOS gelegt.

Die Multiplikation aktiviert »*«.

```
20 20 * . 400 OK
```

Allerdings muß man auch hier aufpassen, nicht den erlaubten Bereich zu verlassen. So ergibt

```
200 200 * . -25534 OK
```



ein falsches Ergebnis (beziehungsweise eine Fehlermeldung). Diesmal liegt es allerdings nicht an den Integerzahlen, sondern an der Tatsache, daß Forth intern nur mit 16 Bit breiten Zahlen rechnet. Da das 16. Bit das Vorzeichen enthält, ist damit der Rechenbereich auf die Zahlen zwischen -32768 und +32767 beschränkt -, zugegebenermaßen ein kleiner Darstellungsbereich. Doch auch hier stehen dem Programmierer alle Türen offen. Theoretisch können Sie mit beliebig breiten Zahlen arbeiten. Von Haus aus erlaubt Forth, entweder auf das Vorzeichen zu verzichten oder aber mit 32 Bit breiten Zahlen zu arbeiten. Beide Fälle setzen allerdings spezielle Wörter voraus. So erhalten Sie bei unserer »verunglückten« Multiplikation mit

```
200 200 * U. 40000 OK
```

doch noch ein vernünftiges Ergebnis. Mit »U.« wird die Zahl, die im TOS steht, als vorzeichenlose Zahl auf dem Bildschirm ausgegeben.

Das Rechnen mit 32-Bit-Zahlen erfordert spezielle Wörter. Diese erkennt man fast immer an einem »D« oder einer »2«, mit der der eigentliche Befehl beginnt. Damit der Textinterpreter solch eine doppelt lange Zahl korrekt erkennt, muß diese mit einem Dezimalpunkt eingegeben werden. Allerdings spielt es dabei keine Rolle, an welcher Stelle dieser Punkt steht.

```
222.222 OK
44.4444 OK
D+ OK
```

Durch »D+« werden die beiden doppelt langen Zahlen im Stack addiert und das Ergebnis im TOS abgelegt. Wie bekommen Sie nun diesen Wert auf den Bildschirm? Sicher nicht mit ».«. Denn damit erhalten Sie nur die niederwertigen 16 Bit der 32-Bit-Zahl ausgegeben. Auch hier ist ein besonderes Ausgabewort notwendig, nämlich »D.«.

```
D. 666666 OK
```

Bislang wurde die Anordung der Zahlen auf dem Stack durch die Reihenfolge der Eingabe festgelegt. Sehr oft besteht jedoch die Notwendigkeit, diese Reihenfolge zu verändern, beziehungsweise einzelne Zahlen auf dem Stack zu kopieren. Auch dazu stehen in Forth eine Reihe von Befehlen zur Verfügung. Die wichtigsten erklären wir Ihnen im folgenden.

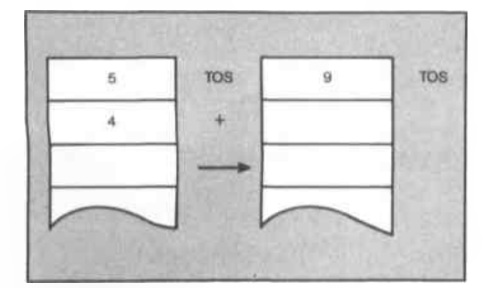


Bild 1. So verändert sich der Stack bei der Addition

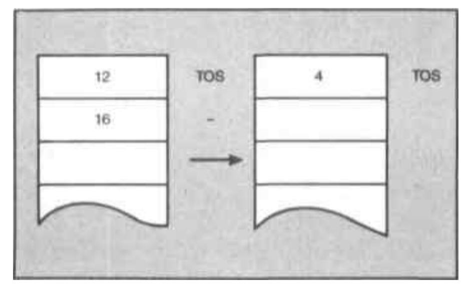


Bild 2. So verändert sich der Stack bei der Subtraktion

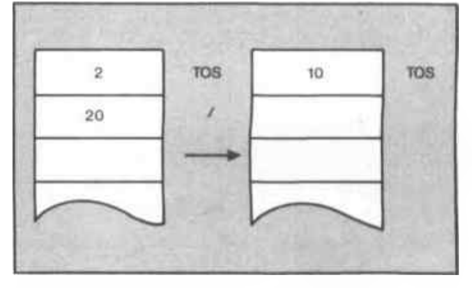


Bild 3. So verändert sich der Stack bei der Division

```
34 DUP OK
. . 34 34 OK
```

»DUP« (Bild 4) kopiert die Zahl im TOS und verschiebt die darunterliegenden um eins nach unten. Die Operation ist beispielsweise immer dann notwendig, wenn eine Zahl im TOS zwar ausgegeben, mit ihr aber noch weiter gerechnet werden soll.

```
67 88 SWAP OK
. . 67 88 OK
```

»SWAP« (Bild 5) vertauscht die beiden obersten Zahlen im Stack. Normalerweise muß ja die zuletzt eingegebene Zahl als erste wieder auf dem Bildschirm erscheinen (LIFO-Prinzip: Last in first out). Das wäre hier die 88 gewesen. »SWAP« hat nun aber die beiden obersten Werte im Stack vertauscht, so daß die 67 im TOS stand und damit auch zuerst ausgegeben wurde.

```
12 33 65 ROT OK
. . . 12 65 33 OK
```

»ROT« läßt die obersten drei Zahlen im Stack einmal gegen den Uhrzeigersinn rotieren. Wie sich dabei der Stack verändert, zeigt am besten Bild 6. Durch »ROT« wird die Zahl von der dritten Stelle im Stack ins TOS gebracht, während die beiden darüberliegenden Werte um eine Position nach unten wandern. Alle Befehle (und noch einige mehr) finden Sie in der Tabelle noch einmal zusammengefaßt.

Normalerweise werden alle Ein- und Ausgaben von Zahlen im Dezimalsystem durchgeführt. Forth ist jedoch in der Lage, beispielsweise die Zahlen in nahezu jedem System auszugeben. Dazu ist lediglich der Inhalt einer einzigen User-Variablen mit dem Namen »BASE« zu ändern. Bei den User-Variablen handelt es sich um Speicherwerte, die wichtige Systemgrößen beinhalten. So enthält zum Beispiel »SO« die Adresse des Stacks, »DP« den Beginn des Wörterbuches und »BASE« den Wert der aktuellen Zahlenbasis.

Der Aufruf einer User-Variablen holt jedoch in Forth nicht den Wert, sondern die Speicheradresse, unter der der Wert zu finden ist. Um den aktuellen Wert von »BASE« zu erfahren, brauchen wir ein Wort (Anweisung) vom Typ: »Gib den Inhalt der Speicherzelle mit der Adresse addr aus«. Solch ein Wort stellt »c@« (manchmal auch »?«) zur Verfügung.

```
BASE C . 10 OK
```

Diese Anweisung bringt den aktuellen Wert der User-Variablen BASE auf den Bildschirm. Um den Wert zu ändern, benötigen wir ein Wort vom Typ: »Lege den Wert a unter der Adresse addr ab«. Dieses Wort lautet »!«. Sowohl der Wert »a« als auch die Adresse »addr« müssen sich auf dem Stack befinden – und zwar in der richtigen Reihenfolge. Nach der Eingabe muß die Adresse (in diesem Fall BASE) im TOS stehen.

```
63 2 BASE ! OK
. 111111 OK
```

Die Zahlen 63 und 2 werden auf den Stack gelegt, dann die Adresse BASE im TOS gespeichert. Das Wort »!« ändert die Ausgabefunktion auf Dualzahlen. Mit ».« erscheint dann die 63 in dualer Form (111111) auf dem Bildschirm. Da wir gerade auf Dualzahlen umgeschaltet haben, muß jetzt auch die Eingabe in dualer Form erfolgen.

```
9 23 ? CAN'T FIND
```

Die Zahlen 9 und 23 sind keine Dualzahlen und deshalb gibt der Compiler eine Fehlermeldung (hier »CAN'T FIND«) zurück. Mit

```
11 BASE ! OK
```

schalten wir auf die Zahlenbasis »3« um (11 dual ist 3 dez). Da es nun etwas kompliziert ist, die richtige Ziffernfolge für Dezimalzahlen zu finden (10 dez = 1010 dual = 101 zur Basis 3), gibt es unter Forth das Wort »DECIMAL«, das immer wieder zum vertrauten Dezimalsystem zurückführt.

```
DECIMAL OK
```

Ein kleines Beispiel zeigt eine beeindruckende Lösung eines Problems, das in vielen anderen Sprachen nur bedeutend umständlicher gelöst werden kann.

```
: DUAL 2 BASE ! ; OK
: DOPPEL
  20 0 DO CR I DECIMAL .
     I DUAL . LOOP ; OK
```

Wir haben zuerst das Wort »DUAL« zur Umschaltung auf Dualzahlen definiert und dann die Anweisung »DOPPEL«. Wie man Wörter bestimmt und auch wie die Schleife »DO ... LOOP« arbeitet, finden Sie in den folgenden Kapiteln. Hier sollen Sie das Programm nur eintippen und starten. Nach dem Aufruf von »DOPPEL« ergibt sich auf dem Bildschirm folgendes Bild:

```
0 0
1 1
2 10
3 11
4 100
5 101
6 110
...
```

(Peter Monadjemi/hg)

Die Erklärung der Darstellung der Befehle finden Sie im letzten Kapitel.

Befehl	Beschreibung
+ (n1 n2 bis n3)	– addiert die beiden obersten Zahlen des Stacks und legt das Ergebnis im TOS ab
– (n1 n2 bis n3)	– subtrahiert n2 von n1 und legt das Ergebnis im TOS ab
/ (n1 n2 bis n3)	– dividiert n1 durch n2 und legt das Ergebnis im TOS ab
MOD (n1 n2 bis n3)	– dividiert n1 durch n2 und legt den ganzzahligen Rest im TOS ab
c@ (a bis n)	– holt ein Byte und speichert es im TOS
? (a bis n)*	– wie »c@«, aber mit Ausgabe auf dem Bildschirm
! (n a usw.)	– speichert Zahl in der Adresse a, die im TOS angegeben ist

Tabelle. Die in diesem Abschnitt neu vorgestellten Befehle und deren »Geschwister«

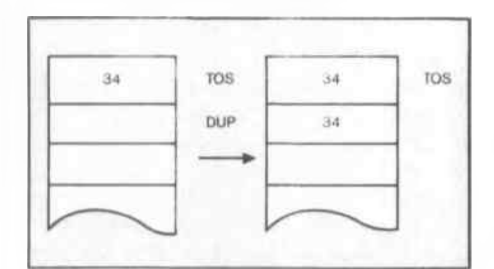


Bild 4. »DUP« verdoppelt die Zahl im TOS

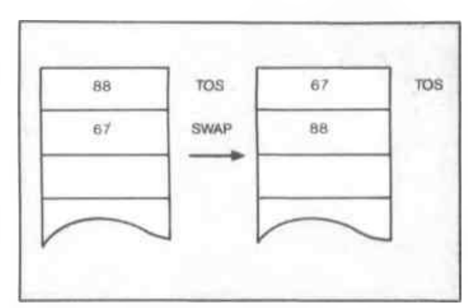


Bild 5. »SWAP« vertauscht den Wert im TOS mit dem darunterliegenden Stackwert

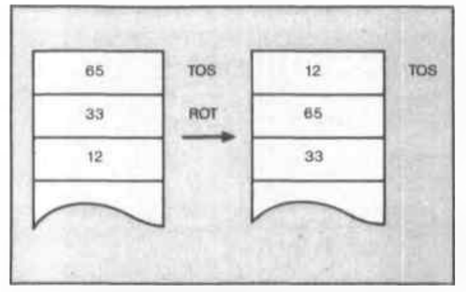


Bild 6. »ROT« bringt den dritthöchsten Wert in den TOS und schiebt die beiden anderen Werte um eins weiter nach unten



Forth lernt dazu

Einen der größten Vorteile von Forth macht der sehr einfache Ausbau seines Wortschatzes aus. Jede Forth-Version kann sein Benutzer mit neuen, selbstdefinierten Anweisungen erweitern.

Anders als beispielsweise in Basic bestehen Forth-Programme nicht aus einer bestimmten Anzahl von Programmzeilen, sondern aus Wörtern. Jedes diese Wörter kann weitere beinhalten, die wiederum neue Wörter enthalten dürfen und so weiter. Dieser extrem modulare Aufbau führt dazu, daß das eigentliche Hauptprogramm letzten Endes aus nur einem einzigen Wort bestehen kann.

Ein solches »Forth-Wort« läßt sich mit einem Unterprogramm in Basic oder einer Prozedur in Pascal vergleichen. Es gibt verschiedene Wege, sich selbst solch eine Anweisung zu definieren. Die einfachste ist die sogenannte »Colondefinition«, bei der das Wort weitere Wörter enthalten darf, die nach Aufruf des neuen Worts ausgeführt werden. Handelt es sich bei diesen Wörtern wiederum um Forth-Wörter, dann spricht man von SECUNDARIEs. Ein Wort, das direkt Maschinencode-Routinen aufruft, bezeichnet man als PRIMITIVE. Die meisten Wörter des Grundwortschatzes von Forth sind SECUNDARIEs.

Wie kann man nun solch ein Wort selbst definieren?

Jede Wortdefinition eines SECUNDARIEs leitet ein unscheinbarer Doppelpunkt »:« ein. Er bewirkt unter anderem, daß das System in den »Compile mode« umschaltet. Das hat zur Folge, daß alle nun folgenden Anweisungen nicht mehr direkt ausgeführt, sondern in das Wörterbuch eingetragen werden.

Unter einem Wörterbuch (englisch Dictionary) wird in Forth ein Speicherbereich verstanden, der den Wortschatz der betreffenden Version beinhaltet. Jede neue Wortdefinition wird nun auch in diesem Wörterbuch verzeichnet. Das Dictionary selbst ist in sogenannte Vokabulare unterteilt. Zwischen verschiedenen Vokabelbereichen schaltet VOKABULARY NAME um.

Durch Aufruf von NAME wird der Vokabelbereich NAME zum CONTEXT-VOKABULAR, das heißt dem aktuellen Vokabular, in dem die Dictionary-Suchläufe zuerst beginnen. Normalerweise ist das Standard-Vokabulary eingeschaltet. Es trägt den Namen Forth.

Wenn wir jetzt ein neues Wort definieren, so wird dies in das »Haupt«-Wörterbuch eingetragen. Durch »WORDS«, »VLIST« oder einem anderen, compilerspezifischen Namen kann man das überprüfen, denn dieser gibt ja den gesamten Inhalt des Wörterbuchs aus. Das Wörterbuch baut sich übrigens in Richtung größer werdender Adressen auf. Das Ende, also den Beginn des freien Arbeitsspeichers, kann man mit »HERE« ins »Top of Stacks« (TOS) laden. Nun aber zurück zu unserem Problem, selbst neue Worte zu definieren.

Anders als in jeder mittelmäßigen Basic-Version fehlt in Forth die Quadratfunktion. Sie ist aber relativ einfach zu definieren. Die Zahl, welche quadriert werden soll, muß zuerst einmal im TOS stehen. Dann kopieren wir sie mit DUP und multiplizieren beide miteinander. Die beiden Wörter, die wir dazu brauchen, kennen Sie schon.

```
4 DUP * . 16 OK
16 DUP * . 256 OK
```

Unsere Überlegung scheint zu stimmen, denn in beiden Fällen wurde die Ausgangszahl quadriert.

Um die Quadratfunktion nun im Wörterbuch zu »verewigen«, erweitern wir es um das Wort »QUADRAT«. Diese Funktion soll bei ihrem Aufruf immer den aktuellen Wert im TOS quadrieren und das Ergebnis dort auch wieder ablegen.

Wie schon erwähnt, leitet ein Doppelpunkt die neue Wortdefinition ein. Danach folgt der Name der neuen Funktion, dann die Befehlsfolge. Ein Semikolon schließt das Ganze ab. Mit

```
: QUADRAT DUP * ; OK
```

haben Sie nun den Wortschatz Ihres Systems bereichert. Bevor Sie das neue Wort aufrufen, müssen Sie aber daran denken, daß die Zahl, die quadriert werden soll, im TOS steht.

```
12 QUADRAT OK
```

berechnet das Quadrat von 12. Das Ergebnis 144 bekommen wir mit

```
. 144 OK
```

auf den Bildschirm. An oberster Stelle im Wörterbuch steht nun das neue Wort. Falls Ihnen die Ausgabeform zu nüchtern ist, dann definieren Sie doch mit

```
: AUSGABE CR
." DIE QUADRATZAHL IST" . ; OK
```

eine Ausgaberoutine. Nach dem Aufruf von AUSGABE erscheint die gerade aktuelle Zahl aus dem TOS. Der Bildschirm sieht dann wie folgt aus:

```
12 QUADRAT AUSGABE
DIE QUADRATZAHL IST 144
```

Um nun die Ausgaberoutine in dem Wort QUADRAT gleich mit aufzurufen, bedarf es einer Neudefinition dieses Wortes. Es gibt zwar auch Wege, bestehende Routinen zu verändern, aber das lassen wir hier beiseite. Also geben wir das Wort schnell noch einmal neu ein.

```
: QUADRAT DUP * AUSGABE ;
? QUADRAT ISN'T UNIQUE
```

Lassen Sie sich durch die Fehlermeldung nicht irritieren. Damit teilt Ihnen der Compiler nur mit, daß es bereits ein Wort mit diesem Namen gab. Durch die neue Definition haben Sie es aber umbenannt. Rufen Sie jetzt QUADRAT auf, und Sie erhalten das gewünschte Ergebnis:

```
5 QUADRAT
DIE QUADRATZAHL IST 25 OK
```

Ein erneuter Blick ins Wörterbuch zeigt, daß jetzt zwei Wörter mit dem Namen QUADRAT existieren. Um ein Wort zu löschen, benutzt man FORGET.

```
FORGET QUADRAT OK
```

Damit bleibt nur noch die Frage zu klären, welche Version der beiden Worte QUADRAT gelöscht wurde. Am einfachsten ist das festzustellen, indem man QUADRAT noch einmal aufruft.

```
10 QUADRAT OK
```

Damit ist klar, daß FORGET die neueste Version unseres Wortes gelöscht hat. Doch damit nicht genug. FORGET löscht nicht nur das betreffende Wort, sondern auch alle anderen, die später definiert wurden. Deshalb sollte man FORGET nur sehr vorsichtig einsetzen.

Der Grund für diese Wirkungsweise von FORGET ist leicht zu verstehen, wenn man sich vorstellt, daß alle Wörter im Wörterbuch durch eine »KETTE« miteinander verbunden sind. Trennen Sie die Kette an einer Stelle (etwa durch FORGET), so sind auch alle Wörter verloren, die bis zu diesem Punkt auf der Kette aufgereiht wurden.

Nicht immer läßt Forth das Löschen von Wörtern so ohne weiteres zu. In den meisten Forth-Versionen ist der Sprachkern geschützt. Eine User-Variable »FENCE« enthält die Adresse, ab der ein Löschen durch FORGET nicht mehr möglich ist.

Noch ein Wort zur Namensgebung. Hier dürfen Sie Ihrer Kreativität freien Lauf lassen, denn als Wortname ist jede beliebige Zeichenkombination erlaubt, die nicht länger als 31 Zeichen ist. Lediglich Leerzeichen dürfen nicht benutzt werden.

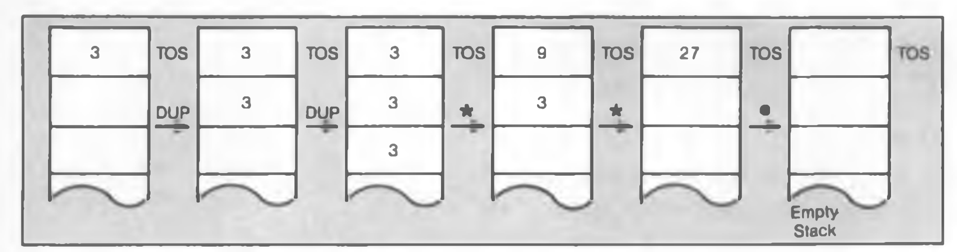
Daß sich in Forth so ziemlich alles um, beziehungsweise neu definieren läßt, zeigt das folgende Beispiel:

```
: 8 6 ; OK
```

Damit wurde der Zahl 8 kurzerhand eine neue »Bedeutung« gegeben. Denn auf einmal erhalten Sie mit

```
8 QUADRAT . 36 OK
```

ein recht merkwürdiges Ergebnis. Forth



So wird der Stack durch das Wort KUBIK verändert

: - Leitet die Definition eines Forth-Wortes ein.
; - Beendet die Definition eines Forth-Wortes.
FORGET - Löscht alle Wörter bis einschließlich dem angegebenen aus dem Wörterbuch.

Die neuen Worte dieses Abschnitts

ist, und das werden Sie noch öfters feststellen, die Sprache der nahezu unbegrenzten Möglichkeiten.

Um Ihnen die Wortbildung noch einmal zu verdeutlichen, definieren wir noch eine zweite Funktion.

```
: KUBIK DUP DUP * * . ;
```

Mit der Funktion KUBIK wird ab sofort die Kubikzahl des Ausgangswerts im TOS auf den Bildschirm ausgegeben. Dazu verdoppelt DUP die Zahl auf dem TOS zweimal und multipliziert dann die beiden obersten Werte jeweils miteinander (»* *«). Der Doppelpunkt und das Semikolon umschließen die Definition.

Mit

```
3 KUBIK 27 OK
```

bekommen wir das gesuchte Ergebnis.

```
100 KUBIK 16960 OK
```

ist jedoch ein etwas seltsames Ergebnis. Hier müssen wir uns wieder den Wertebereich unserer Zahlen ins Gedächtnis zurückrufen. Denn 1000000 können wir mit unserem Zahlenbereich zwischen -32768 und 32767 nicht darstellen.

Nachdem Sie nun eine ganze Menge über die Wortdefinition und das Forth-Wörterbuch gelernt haben, ist es an der Zeit, eine Zusammenfassung durchzuführen:

- Jede Wortdefinition wird in das Wörterbuch eingetragen. Dabei ist der hier besprochene Doppelpunkt nicht der einzige Weg, eine Wortdefinition vorzunehmen.
- Ein Eintrag in das Wörterbuch hat zur Folge, daß das jeweilige Wort Bestandteil des Wortschatzes wird, und somit auch, genauso wie die Worte aus dem Grundwortschatz, aufgerufen werden kann.
- Durch FORGET NAME wird die letzte Definition NAME gelöscht, sowie alle danach durchgeführten Definitionen.

(Peter Monadjemi/hg)

Forth, entscheiden Sie sich!

UPN, Rechnen mit dem Stack und Wortdefinitionen sind nach den letzten Seiten kein Problem mehr für Sie. Aber ein Computer-Programm muß auch Entscheidungen treffen können.

Wie in fast allen höheren Programmiersprachen, können auch in Forth Entscheidungen in der Form »Führe eine Anweisung nur dann aus, wenn ein Vergleich positiv ausfällt« bearbeitet werden. Forth stellt dazu zwei Konstruktionen zur Verfügung. Zum einen »IF ... ENDIF« und zum andern »IF ... ELSE ... ENDIF«. Die zweite Anweisung bearbeitet den Teil zwischen IF und ELSE, wenn der Vergleich positiv ist. Ist er negativ, dann wird der Teil ausgeführt, der zwischen ELSE und ENDIF steht. Fast alle Forth-Dialekte erlauben anstelle von ENDIF auch THEN, wenn auch der erste Weg die sinnvollere Bezeichnung darstellt.

Ein Beispiel verdeutlicht die Arbeitsweise von IF ... ENDIF:

```
: TEST 9 >IF ."ZU GROSS !"
  ENDIF ; OK
```

»TEST« prüft, ob die Zahl im TOS (also die zuletzt eingegebene Zahl) größer als 9 ist. In diesem Fall wird der Kommentar »ZU GROSS !« ausgegeben.

```
4 TEST OK
11 TEST ZU GROSS OK
```

Wenn Sie sich nun einmal das zugehörige Stack-Diagramm (Bild 1) anschauen, dann wird Ihnen der Mechanismus der IF-ENDIF-Anweisung schnell klar. Um einen Vergleich durchzuführen, müssen sich zunächst einmal beide Zahlen im Stack befinden. Der Vergleichsoperator »>« holt beide Zahlen vom Stack, führt den Vergleich aus und legt für den Fall, daß er positiv ausfällt eine »1« und für den Fall, das er negativ ausfällt eine »0« im TOS ab. Von diesem Flag (deutsch: Flagge oder Signal) hängt es ab, ob die zwischen IF und ENDIF stehenden Anweisungen ausgeführt werden oder nicht.

Bei der IF-ELSE-ENDIF-Anweisung dagegen werden für den Fall, daß der Vergleich negativ ausfällt, die Anweisungen zwischen ELSE und ENDIF ausgeführt. Alle erlaubten Vergleichsoperatoren zeigt Tabelle 1.

Wenn Sie die Vergleichsoperatoren durchgehen, dann werden Sie sicher den Vergleich »ungleich« vermissen. Er läßt sich aber leicht durch die Wortfolge »= NOT« ersetzen. Das Flag, das bei »=« im TOS abgelegt wird, invertiert dann »NOT« und wir haben unser Ziel erreicht.

Zwei Punkte müssen Sie aber noch bedenken, wenn die Anweisung für eine Entscheidung dienen soll. Sie darf nie im Direktmodus, sondern nur innerhalb einer Wortdefinition stehen. Außerdem ist noch zu beachten, daß die Zahlen, mit denen man den Vergleich durchgeführt hat, anschließend vom Stack verschwunden sind. Wollen Sie weiter mit diesen Werten arbeiten, dann müssen Sie sie zuvor kopieren.

Nun zu einem anderen Thema: Die Stärke eines Computers liegt in seiner Fähigkeit, bestimmte Anweisungen beliebig oft sehr schnell zu wiederholen. Während im normalen Basic hierfür nur die FOR-NEXT-Schleife vorhanden ist, kennt Forth insgesamt vier verschiedene Anweisungen. Allen gemeinsam ist, daß sie nur in Wörtern (also nicht direkt) benutzt werden dürfen. Die einfachste Wiederholfunktion ist »DO LOOP«:

```
: SCHLEIFE 10 0 DO I .
          LOOP ; OK
```

Als Ergebnis bekommen wir

```
SCHLEIFE 0 1 2 3 4 5 6 7 8 9 OK
```

Zwei Besonderheiten fallen an der Schleifenkonstruktion sofort auf:
- Zuerst wird der End- und dann der Anfangswert übergeben.
- Das Wort »I« holt den Schleifenwert, der die bisherige Anzahl der Durchläufe angibt, in den TOS.

Die Bedeutung des Wortes »DO« besteht darin, zum einen die Stelle zu markieren, zu der nach »LOOP« unter Umständen zurückgekehrt wird, und dient zum anderen dazu, die beiden Schleifenwerte (Start und Ende) auf einen weiteren Stack zu transferieren. Von diesem war bisher noch nicht die Rede, da er für den Anfänger keine praktische Bedeutung hat. Die Aufgabe dieses »RETURN STACK« besteht

darin, wichtige Adressen bei der Ausführung eines Wortes zu verwalten. Und auch die Schleifenwerte einer DO-LOOP-Anweisung werden hier gespeichert. Damit ist auch die Bedeutung des Wortes »I« zu verstehen. Diese Anweisung holt den momentanen Wert des »Top Of Return Stack« in den TOS.

Die DO-LOOP-Anweisung gehört zu den sogenannten »definierten Schleifen«, da die Anzahl der Durchläufe von vornherein fest steht. Ganz anders ist das bei »BEGIN ... UNTIL«:

```
: TEST BEGIN 1 + DUP .
       DUP 100 =
       UNTIL ." FERTIG !" ; OK
```

Hier steht die Anzahl der Durchläufe nicht von vornherein fest. Sie hängt vielmehr von einer Bedingung ab. In unserem Beispiel wird durch »=« geprüft, ob der Inhalt des TOS bereits den Wert 100 erreicht hat. Ist das der Fall, so wird im TOS eine 1 als Flag abgelegt. Daran erkennt das Wort UNTIL, daß eine Anweisung abzubrechen ist.

Für den Fall, daß der Inhalt des TOS 100 noch nicht erreicht hat, wird durch »=« eine 0 im TOS abgelegt und anschließend werden alle Anweisungen, die zwischen BEGIN und UNTIL liegen, ein weiteres Mal wiederholt. Somit handelt es sich bei dieser Anweisung um eine vom Typ: »Wiederhole so lange, bis eine bestimmte Bedingung wahr ist.«

Falls Ihnen dieser Befehl immer noch zu undurchsichtig ist, so nehmen Sie ein Blatt Papier und zeichnen Sie die Stackbelegung bei dem Wort TEST auf. Denken Sie daran, daß sowohl durch ».«, als auch durch »=« der Inhalt des TOS vom Stack verschwindet, wenn man ihn nicht vorher kopiert hat.

Noch ein Beispiel für diese Befehlsfolge:

```
: UEBUNG BEGIN CR
  ." DRUECKE EINE TASTE" KEY
  65 = UNTIL ; OK
```

Durch KEY wird ein Zeichen von der Tastatur gelesen (ähnlich GET in Basic) und der dazugehörige ASCII-Wert im TOS abgelegt. Dieser wird daraufhin mit 65 verglichen (ASCII-Wert von A ist 65). Haben Sie tatsächlich A eingegeben, so bricht der Programmlauf ab, andernfalls wird er ein weiteres Mal durchgeführt.

Ähnlich wie bei BEGIN-UNTIL liegen die Verhältnisse bei der BEGIN-WHILE-REPEAT-Anweisung mit dem Unterschied, daß die Ausführungs-Bedingung bereits vor dem WHILE stehen muß. Die eigentliche Anweisung befindet sich zwischen WHILE und REPEAT. Ist die Bedingung nicht erfüllt, so wird die Anweisung gar nicht erst ausgeführt.

```
: TASTE BEGIN KEY 65 = 0=
 WHILE ." VERSUCH'S NOCHMAL "
 UNTIL ." NA ENDLICH !" ; OK
TASTE B VERSUCH'S NOCHMAL OK
TASTE A NA ENDLICH ! OK
```

KEY bringt wieder den ASCII-Code der gerade gedrückten Taste in den TOS. »65 =« prüft diesen Wert auf Code 65 (für A). Da in diesem Fall im TOS eine »1« abgelegt und dadurch die Anweisung zwischen WHILE und UNTIL nicht ausgeführt werden würde, wird der Inhalt des TOS (das Flag) mit »0=« invertiert. Wenn im TOS eine 0 liegt, dann verändert »0=« diese in eine 1.

Zum Abschluß dieses Kapitels noch eine Wiederholungsanweisung, bei der Sie sich keine Gedanken um ein Abbruchkriterium machen müssen. Denn bei »BEGIN-AGAIN« handelt es sich um eine Endlosanweisung.

```
: NONSTOP BEGIN 40 EMIT
          AGAIN ; OK
```

Nach dem Aufruf von NONSTOP produziert Ihr Computer so lange Klammeraffen, bis Sie den Strom abschalten oder einen Reset durchführen.

Variablen, ohne die kaum ein Basic- oder Pascal-Programm auskommt, haben wir bisher eher beiläufig erwähnt. Das liegt daran, daß Forth (anders als Basic und Pascal) stackorientiert ist. Damit ist gemeint, daß sich alle Zahlenoperationen auf dem Stack abspielen. Demnach könnte man streng genommen auf die Variablen völlig verzichten. In bestimmten Fällen haben diese aber doch ihre Bedeutung. Denn zum einen ist die Kapazität des Stacks begrenzt, und zum anderen ersparen Sie sich durch die Verwendung von Variablen, beziehungsweise Konstanten, allzu umständliche Stack-Operationen.

Um mit Konstanten oder Variablen zu arbeiten, müssen Sie diese (ähnlich Pascal) zuerst einmal definieren und ihnen einen Anfangswert zuweisen. Dies geschieht durch die Definitionswörter »CONSTANT« und »VARIABLE«.

```
0 VARIABLE ZAHL OK
```

Damit haben wir eine Variable auf den Namen Zahl definiert. Beim Aufruf von Zahl erhalten Sie nun aber nicht deren Wert, sondern lediglich die Adresse, unter der dieser Wert im Speicher zu finden ist.

```
ZAHL . 12345 OK
```

Um den eigentlichen Wert zu erfahren, benutzen wir den schon bekannten Befehl »C@« beziehungsweise »?«. Seine Bedeutung war: »Hole den Inhalt der Speicherzelle, deren Adresse im TOS liegt.« Mit

```
4 ZAHL ! OK
```

weisen wir unserer Variablen ZAHL den Wert 4 zu. Zuerst laden wir die 4 und die Adresse von ZAHL auf den Stack. Mit

```
ZAHL   4 OK
```

testen wir, ob die neue Zahl im Speicher abgelegt wurde.

Ein wenig anders schaut es mit den Konstanten aus. Hier bringt der Aufruf der Konstanten mit Namen ihren Wert direkt in den TOS:

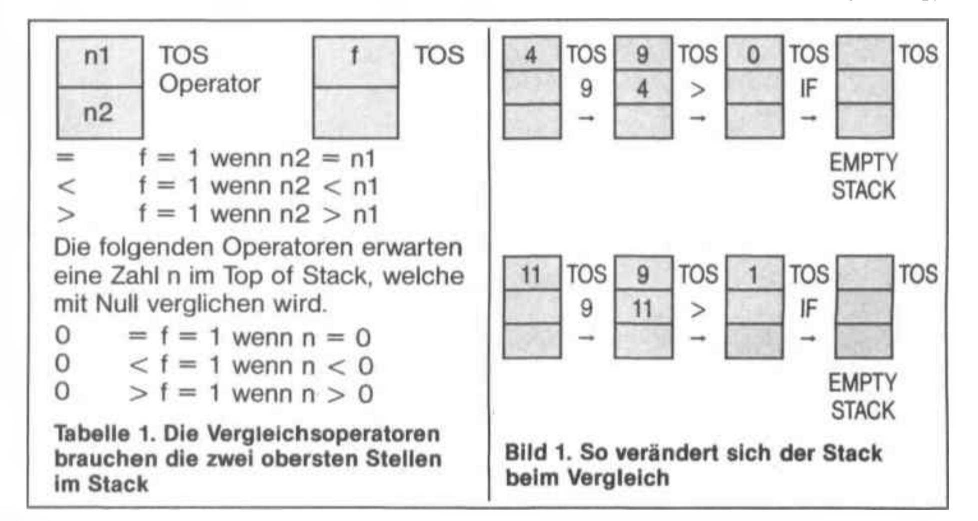
```
45 CONSTANT WERT OK
WERT . 45 OK
```

Auch der Wert einer Konstanten läßt sich ändern, wenn Sie die Adresse kennen. Als wir weiter vorne die Zahlenbasis veränderten, griffen wir auf solch eine Variable zurück. Zum Abschluß noch ein kleines Zählprogramm, das im Hexadezimalsystem bis 100 zählt:

```
: HZAEHL
  100 0 DO HEX I . CR LOOP
  DECIMAL ; OK
```

Programmieren können Sie in Forth jetzt natürlich noch lange nicht. Aber Sie kennen die Grundzüge und es steht Ihnen nichts im Wege, sich mit offenen Augen in das Abenteuer Forth zu stürzen. In der folgenden Tabelle 2 finden Sie diesmal nicht die neu besprochenen Befehle, sondern alle diejenigen, die Ihr Forth-System haben sollte, aber nicht haben muß. Die Erklärung der Befehlsworte spornt Sie vielleicht an, die Anweisungen auszuprobieren und in eigenen Programmen zu verwenden. Die Listings in diesem Heft zeigen Ihnen weiter, wie man Forth-Programme entwickelt und realisiert.

(Peter Monadjemi/hg)



n1	TOS Operator	f	TOS
n2			

=	f = 1 wenn n2 = n1
<	f = 1 wenn n2 < n1
>	f = 1 wenn n2 > n1

Die folgenden Operatoren erwarten eine Zahl n im Top of Stack, welche mit Null verglichen wird.

0 =	f = 1 wenn n = 0
0 <	f = 1 wenn n < 0
0 >	f = 1 wenn n > 0

Tabelle 1. Die Vergleichsoperatoren brauchen die zwei obersten Stellen im Stack

Bild 1. So verändert sich der Stack beim Vergleich

Wort	Beschreibung	Stack-Relation	Wort	Beschreibung	Stack-Relation
!	Speichert eine Zahl in der Adresse an oberster Stack-Position	n a -	2@	Holt eine doppelt genaue Integer aus der Adresse	a - d
#	Dient bei der Zahlenausgabe mit Maske für die Zifferndarstellung vorzeichenloser doppelt genauer Integers	$d_1 - d_2$	2ARRAY	Definiert einen zweidimensionalen Array	$n_1\ n_2$ -
# >	Beendet die maskierte Zahlenausgabe	d - a n	2CONSTANT	Definiert eine doppelt genaue Konstante	d -
#IN	Fordert zur Eingabe einer einfach genauen Integer auf	- n	2DARRAY	Definiert einen doppelt genauen Integer-Array	$n_1\ n_2$ -
#S	Wandelt bei der Zahlenausgabe mit Maske Ziffernzeichen in den ASCII-Code um	d -	2DROP	Entfernt die oberste doppelt genaue Integer vom Stack	d -
$!	Dient zum Speichern von Strings	a_s a -	2DUP	Dupliziert die oberste doppelt genaue Integer auf dem Stack	d - d d
$"	Vereinbart einen String im Arbeitsspeicher	- a	OVER	Dupliziert die zweite doppelt genaue Integer auf dem Stack an oberste Stack-Position	$d_1\ d_2 - d_1\ d_2 d_1$
$-TB	Entfernt nachlaufende Blanks vom String		2ROT	Rotiert die dritte doppelt genaue Integer an oberste Stack-Position	$d_1\ d_2\ d_3 - d_2\ d_3\ d_1$
$.	Druckt einen String	a -	2SWAP	Vertauscht die obersten beiden doppelt genauen Integers	$d_1\ d_2 - d_2\ d_1$
$ARRAY	Vereinbart einen String-Array	$n_1\ n_2$ -	2VARIABLE	Vereinbart eine doppelt genaue Variable	
$COMPARE	Vergleicht String-Variable	$a_1\ a_2$ - n	:	Leitet die Definition eines FORTH-Wortes ein	
$CONSTANT	Vereinbart eine String-Konstante		;	Beendet die Definition eines FORTH-Wortes	
$VARIABLE	Vereinbart eine String-Variable		<	Wird »wahr« falls $n_1 < n_2$	$n_1\ n_2$ - f
$XCG	Vertauscht die Werte in String-Variablen	$a_1\ a_2$ -	< #	Leitet die Zahleneingabe mit Maske ein	
'	Liefert die Adresse des nächsten Wortes im Eingabestrom	- a	< =	Wird »wahr«, falls n_1 kleiner oder gleich n_2 ist	$n_1\ n_2$ - f
(	Leitet einen Kommentar ein		< >	Wird »wahr«, falls n_1 ungleich n_2 ist	$n_1\ n_2$ - f
*	Liefert das Produkt zweier Zahlen	$n_1\ n_2$ - n	<CMOVE	Dupliziert n Speicherwörter beginnend bei a_1 an der Adresse a_2; Übertragung beginnt bei der höchstwertigen Adresse	$a_1\ a_2$ n -
*/	Multipliziert n_1 mit n_2 und dividiert das doppelt genaue Produkt durch n_3	$n_1\ n_2\ n_3$ - n	=	Ist »wahr«, falls n_1 gleich n_2 ist	$n_1\ n_2$ - f
*/MOD	Ähnlich wie */; liefert jedoch auch den Rest	$n_1\ n_2\ n_3 - n_r\ n_q$	> =	Ist »wahr«, falls n_1 größer oder gleich n_2 ist	$n_1\ n_2$ - f
+	Liefert die Summe zweier Zahlen	$n_1\ n_2$ - n	<IN	Enthält die Startposition für die Untersuchung des Eingabestroms	- a
+!	Inkrementiert den gespeicherten Wert	n a -	>R	Überträgt eine Integer auf den Kontroll-Stack; benötigt entsprechendes R>	n -
+LOOP	Inkrementiert eine Schleifenvariable	n -	?DUP	Dupliziert die oberste einfach genaue Integer, es sei denn, diese ist gleich 0	n - n n
,	Compiliert n ins Wörterbuch	n -	@	Holt die an der Adresse gespeicherte einfach genaue Integer	a - n
-	Subtrahiert n_2 von n_1	$n_1\ n_2$ - n	ABS	Ersetzt die oberste einfach genaue Integer durch ihren Absolutbetrag	$n_1 - n_2$
-TRAILING	Aktualisiert den Zeichenzähler	a n_1	ALLOT	Erweitert den Speicherbereich einer Variablen um n Byte	n -
.	Gibt eine Zahl aus	n -	AND	Bitweises logisches AND	$n_1\ n_2 - n_3$
."	Gibt Text aus		ARRAY	Vereinbart einen Array	
.R	Gibt die Zahl n_1 im Datenfeld n_2 aus	$n_1\ n_2$ -	ASC	Legt den ASCII-Wert des ersten Zeichens in dem String, der bei a beginnt, auf den Stack	a -
/	Dividiert n_1 durch n_2	$n_1\ n_2$ - n	BASE	Enthält die Ein-/Ausgaberadix	a - n
/MOD	Division mit Quotient und Rest	$n_1\ n_2$ -	BEGIN	Leitet eine Schleife ein	- a
0<	»Wahr« falls n < 0	n - f	BLANK	Füllt Speicherbereiche mit Leerzeichen	a n -
0=	»Wahr« falls n = 0	n - f	BLK	Enthält die Adresse des Blockpuffers für den Eingabestrom	- a
0>	»Wahr« falls n > 0	n - f	BLOCK	Überträgt den Block n von der Diskette in den Arbeitsspeicher und legt dessen Startadresse auf den Stack	n - a
1+	Inkrementiert den obersten Stack-Eintrag um Eins	n - n_1			
1-	Dekrementiert den obersten Stack-Eintrag um Eins	n - n_1			
16*	Multipliziert den obersten Stack-Eintrag mit 16	n - n_1			
2!	Speichert eine doppelt genaue Integer	d a -			
2$ARRAY	Definiert einen zweidimensionalen String-Array	$n_1\ n_2\ n_3$ -			
2*	Multipliziert die oberste Integer mit 2	n - n_1			
2+	Addiert 2 auf die oberste Integer	n - n_1			
2-	Subtrahiert 2 von der obersten Integer	n - n_1			
2/	Dividiert die oberste Integer durch 2	n - n_1			

Wort	Beschreibung	Stack-Relation
BUFFER	Wie BLOCK, die Daten werden jedoch nicht übertragen	n – a
C!	Speichert das niedrigwertige Byte einer einfach genauen Integer	n a –
C@	Holt ein Byte und speichert es als einfach genaue Integer	a – n
CASEND	Beendet eine CASE-Anweisung	
CHR$	Wandelt eine ein Byte lange Integer in ihre ASCII-Darstellung um; das Ergebnis steht im temporären Arbeitsbereich, dessen Adresse auf den Stack gelegt wird	c – a
CMOVE	Überträgt n Bytes von Adresse 1 nach Adresse 2; die Übertragung beginnt bei den niedrigwertigen Adressen	a_1 a_2 n –
COMPILE	Nimmt einen Wert in die Wortdefinition mit auf	
CONSTANT	Vereinbart eine Konstante mit dem Wert n	n –
CONTEXT	Enthält die Adresse des Kontext-Vokabulars	– a
COUNT	Legt die Anfangsadresse des Strings und den String-Zähler auf den Stack	a – a_1 n
CR	Sendet einen Zeilenvorschub	
CREATE	Richtet einen Wörterbucheintrag ein	
CRT	Lenkt die Ausgabe auf den Bildschirm	
CURRENT	Enthält die Adresse des aktuellen Wörterbuches	– a
D#IN	Fordert zur Eingabe einer doppelt genauen Integer auf	– d
D*	Multipliziert doppelt genaue Integers	d_1 d_2 – d_p
D*/	Multipliziert d_1 mit d_2 und dividiert das vierfach genaue Produkt anschließend durch d_3	d_1 d_2 d_3 – d
D*/MOD	Wie D*/; liefert aber auch den Rest	d_1 d_2 d_3 d_r d_q
D+	Addiert zwei doppelt genaue Zahlen	d_1 d_2 – d
D-	Subtrahiert zwei doppelt genaue Zahlen (d_1 minus d_2)	d_1 d_2 – d
D/	Liefert den Quotienten von d_1 und d_2	d_1 d_2 – d
D/MOD	Wie D/, liefert aber auch noch den Rest	d_1 d_2 – d_r d_q
D.	Gibt eine doppelt genaue Integer aus	d –
D.R	Gibt eine doppelt genaue Integer in einem n Zeichen langen Datenfeld aus	d n –
E	Bearbeitet den Block, der vom Inhalt von SCR bestimmt wird	
EDIT	Bearbeitet Block n; n wird in SCR gespeichert	n –
ELSE	Für Programmverzweigungen	c –
EMIT	Gibt ein Zeichen aus	
EMPTY-BUFFERS	Markiert alle Puffer als leer	
ERASE	Setzt n aufeinanderfolgende Byte auf den Wert 0, beginnend mit der Adresse a	a n –
EXECUTE	Führt den Wörterbucheintrag aus, dessen Adresse auf dem Stack liegt	a –
EXIT	Beendet die Programmbearbeitung	
EXPECT	Liest Zeichen in den Arbeitsspeicher ein, beginnend bei Adresse a, wobei maximal n Zeichen oder bis zum ersten Return gelesen wird	a n –
FILL	Belegt n aufeinanderfolgende Speicherwörter (beginnend bei Adresse a) mit dem ASCII-Wert n_c	a n n_c –
FIND	Sucht die Adresse des nächsten Wortes im Eingabestrom	– a
FLUSH	Speichert die markierten Puffer auf Diskette	
FORGET	Löscht alle Wörter bis einschließlich dem angegebenen aus dem Wörterbuch	
FORTH	Name des Hauptwörterbuches	
D0=	Ist »wahr«, wenn der doppelt genaue Wert gleich 0 ist	d – f
D<	Ist »wahr«, wenn d_1 kleiner d_2 ist	d_1 d_2 – f
DABS	Liefert den Absolutwert einer doppelt genauen Integer	d_1 – d_2
DARRAY	Vereinbart einen Array mit doppelt genauen Integers	n –
DECIMAL	Setzt die Zahlenbasis auf 10	
DEFINITIONS	Macht den Kontext-Wortschatz zum aktuellen Wortschatz	
DEPTH	Liefert die Stack-Tiefe in Einheiten von einfach genauen Integers	– n
DMAX	Liefert die größere von zwei doppelt genauen Integers	d_1 d_2 – d
DMIN	Liefert die kleinere von zwei doppelt genauen Integers	d_1 d_2 – d
DNEGATE	Dreht das Vorzeichen einer doppelt genauen Integer um	d – -d
DO	Leitet eine Schleife ein	n_1 n_2 –
DRDSECS	Liest Diskettensektoren	a n_1 n_2 n_3 n_4 – nf
DROP	Entfernt die oberste einfach genaue Integer vom Stack	n –
DUP	Dupliziert die oberste einfach genaue Integer	n – n n
DWTSECS	Schreibt Diskettensektoren	a n_1 n_2 n_3 n_4 – nf
HERE	Liefert die Adresse des nächsten verfügbaren Wörterbuch-Bytes	– a
HEX	Umwandlung der Zahlenausgabe in Hexadezimaldarstellung	
HOLD	Zur Einfügung von Zeichen bei der Zahlenausgabe mit Maske	c –
I	Legt den Schleifenindex auf den Stack	– n
I'	Legt den Testwert der Schleife auf den Stack	– n
IF	Für Programmverzweigungen	f –
IMMEDIATE	Schaltet von Compilierung in Ausführung um	n_1 n_2 –
INDEX	Gibt die erste Zeile von n_2 Blocks aus, beginnend mit Block n_1	n_1 n_2 –
J	Liefert den Index der dynamisch übernächsten Schleife auf den Stack	– n
KEY	Legt den ASCII-Code des nächsten Eingabezeichens auf den Stack	– n
L	Gibt den Block aus, dessen Nummer in SCR gespeichert ist	
LEAVE	Beendet eine Schleife	
LEFT$	Überträgt die n ersten Zeichen des Strings, der bei a beginnt, in den temporären Arbeitsbereich	a n – a_1

Tabelle 2. Der Befehlssatz, den Ihr Forth-System haben sollte

Wort	Beschreibung	Stack-Relation
LEN	Legt die Länge eines Strings auf den Stack	a – n
LIST	Gibt den Block n aus und legt n in SCR ab	n –
LITERAL	Nimmt den Stack-Wert, ohne ihn zu interpretieren, in die Compilation mit auf	
LOAD	Lädt den Block n	n –
LOADS	Lädt n_2 Block, beginnend mit n_1	n_1 n_2 –
LOOP	Inkrementiert den Schleifenindex	
M*	Doppelt genaues Produkt zweier einfach genauer Integers	n_1 n_2 – d
M*/	Multipliziert d_1 mit n_2 und speichert das Produkt als dreifach genaue Integer, welche dann durch n_3 dividiert wird; der Quotient ist doppelt genau	d_1 n_2 n_3 – d_2
M+	Gemischte Addition	d_1 n – d_2
M–	Gemischte Subtraktion	d_1 n – d_2
M/	Gemischte Division	d n_1 – n_2
M/MOD	Wie M/, außer daß sowohl Quotient als auch Rest geliefert werden	d n_1 – d_r n_q
MAX	Liefert den größeren von zwei Werten	n_1 n_2 – n
MID$	Überträgt an die Adresse a_1 einen n_2 Zeichen langen Teil-String, der ab der $n_1$ten Zeichenposition des Strings a beginnt	a n_1 n_2 – a_1
MIN	Liefert den kleineren von zwei Werten	n_1 n_2 – n
MOD	Liefert den Rest der Division von n_1/n_2	n_1 n_2 – n
MOVE	Verschiebt n 16 Byte lange Speicherwörter, beginnend bei a_1, nach a_2	a_1 a_2 n –
MYSELF	Erlaubt rekursive Aufrufe	
NCASE	Leitet eine CASE-Anweisung ein	n –
NEGATE	Ersetzt eine Zahl durch die negative Zahl mit dem gleichen Betrag	n – –n
NOT	Negiert ein Flag	f_1 – f_2
OCTAL	Setzt die Ein-/Ausgabebasis für Zahlen auf das Oktalsystem	
OTHERWISE	Allgemeiner Ausgang in CASE-Anweisungen	
OVER	Dupliziert die zweite Zahl an oberste Stack-Position	n_1 n_2 – n_1 n_2 n_1
PAD	Enthält die Anfangsadresse des temporären Arbeitsbereichs	– a
PAGE	Löscht den Bildschirm	
PCRT	Legt die Ausgabe sowohl auf Bildschirm als auch auf Drucker	
PRINT	Legt die Ausgabe nur auf den Drucker	
QUERY	Für Zeicheneingabe	
QUIT	Löscht den Return-Stack	
R>	Überträgt die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	– n
R@	Dupliziert die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	– n
RANDOMIZE	Initialisiert den Zufallsgenerator	
REPEAT	Für die Programmierung von Schleifen	
RIGHT$	Überträgt die n letzten Zeichen des Strings a in den temporären Arbeitsbereich, liefert dessen Adresse	a n – a_1

Tabelle 2. Der Befehlssatz, den Ihr Forth-System haben sollte (Schluß)

Wort	Beschreibung	Stack-Relation
RN1	Erzeugt eine Zufallszahl und speichert sie in SEED	
RND	Erzeugt eine Zufallszahl zwischen 1 und n_1	n_1 – n_2
ROLL	Legt die n-te einfach genaue Integer auf dem Stack an oberste Stack-Position	n_0 – n_n
ROT	Befördert die dritte einfach genaue Integer an oberste Stack-Position	n_1 n_2 n_3 – n_2 n_3 n_1
SAVE-BUFFERS	Markiert alle Puffer für nachfolgende Sicherungen	
SCR	Enthält die Adresse des zuletzt bearbeiteten Blockpuffers	– a
SIGN	Fügt den ASCII-Code des Minuszeichens bei Zahlenausgabe mit Maske ein, falls n negativ ist	n –
SPACE	Gibt ein Leerzeichen aus	
SWAP	Vertauscht die beiden obersten Stack-Einträge	n_1 n_2 – n_2 n_1
THEN	Bei Programmverzweigungen benötigt	
TYPE	Gibt n Zeichen beginnend ab der Adresse a aus	a n –
U*	Vorzeichenlose Integermultiplikation	u_1 u_2 – u
U.	Gibt eine vorzeichenlose Integer aus	u –
U.R	Gibt eine vorzeichenlose Integer in einem n Stellen breiten Datenfeld aus	u n –
U/MOD	Vorzeichenlose Division mit doppelt genauem Dividenden, liefert Quotienten und Rest	u_d u_1 – u_r u_q
U<	»Wahr« falls u_1 kleiner u_2 ist (vorzeichenlose Integers)	u_1 u_2 –
UNTIL	Für die Programmierung von Schleifen	f –
UPDATE	Markiert alle Blockpuffer als gesichert	
VARIABLE	Definiert eine Variable	
VOCABULARY	Für die Vereinbarung eines neuen Wortschatzes	
WHILE	Für die Programmierung von Schleifen	f –
WORD	Liest Zeichen aus dem Eingabestrom; Trenner ist Zeichen mit ASCII-Code n	n – a
XOR	Bitweises exklusives ODER	n_1 n_2 – n
Y/N	Fragt nach Y oder N; N liefert Wahrheitswert »wahr«	– f
[	Beendet Compilierung und leitet Ausführung ein; wird in Wortdefinition benötigt	
[COMPILE]	Bewirkt, daß ein Wort mit dem Status IMMEDIATE compiliert wird	
]	Beendet Ausführung und fährt mit der Compilierung fort	

Tabelle aus »Der Einstieg in Forth«, Markt&Technik Verlag AG, ISBN 3-89090-085-2

Die Buchstaben der Stack-Relationsspalte bedeuten:

- a = Adresse
- c = ASCII-Code
- d = doppeltgenaue Zahl
- f = Flag
- n = ganze Zahl
- r = Rest (bei Division)
- q = ganzzahliges Ergebnis (bei Division)

Forth zum Abtippen

Gelerntes will auch geübt werden. Wer noch keinen Forth-Interpreter hat, der findet hier einen, der nichts kostet. Einfach eintippen, RUN eingeben und mit der ENTER-Taste starten.

Nachdem Sie sich jetzt durch viele Seiten Forth hindurchgekämpft haben, wollen Sie Ihre Forth-Programme zum Laufen bringen. Doch gleich einen Compiler kaufen, das muß nicht sein. Testen Sie erst einmal Ihr Interesse mit diesem kostenlosen Basic-Forth-Interpreter.

»Basic-Forth V.4« ist vollständig in Basic geschrieben und läuft auf beinahe jedem Computer. Spezielle Befehle wurden fast völlig weggelassen und erklären sich, wenn, durch ihre Anweisungen. Also einfach eingetippt und schon beginnt Ihr Forth-Vergnügen.

Die Profis unter Ihnen werden jetzt sicher schmunzeln. Forth-Interpreter in Basic – da geht doch der ganze Geschwindigkeitsgewinn in die Binsen. Richtig, aber mit Basic-Forth sollen Sie nicht professionell programmieren, sondern ausprobieren. Und da ist Basic zum Eingeben eben die leichteste Programmiersprache. Wenn dann erst einmal Interesse an dieser Sprache geweckt ist, können Sie immer noch auf einen echten Forth-Compiler umsteigen. (hg)

```
1 CLS
2 DIM S(80),R(80),L(80),LO(80),I$(800)
3 DIM B$(80)
4 PRINT " BASIC-FORTH     V.4"
20 REM
24 ON ERROR GOTO 29
28 GOTO 30
29 PRINT A$,"?"
30 M=0
32 N=0
60 K=1
62 INPUT I$
63 I$=I$+" "
64 L1=0
70 L(K)=L1
72 LO(K)=LEN(I$)
74 L1=LO(K)
100 IF N<0 THEN GOTO 106
104 GOTO 110
106 PRINT "STACK EMPTY"
108 GOTO 30
110 L(K)=L(K)+1
112 IF L(K)>LO(K) THEN GOTO 132
114 B$=MID$(I$,L(K),1)
116 IF B$=" " THEN GOTO 110
118 A$=B$
120 L(K)=L(K)+1
122 B$=MID$(I$,L(K),1)
124 IF B$=" " THEN GOTO 130
126 A$=A$+B$
128 GOTO 120
130 GOTO 200
132 IF K<2 THEN GOTO 60
134 K=K-1
135 I$=MID$(I$,1,LO(K))
136 L1=LO(K)
138 GOTO 110
200 REM    DICTIONARY
300 IF A$<>"SQUARE" THEN GOTO 310
302 B$="DUP * "
304 I$=I$+B$
306 K=K+1
308 GOTO 70
310 IF A$<>"CUBE" THEN GOTO 320
312 B$="DUP SQUARE * "
314 I$=I$+B$
316 K=K+1
318 GOTO 70
320 IF A$<>"TEST" THEN GOTO 330
322 B$="DO PI 10 / R@ * SIN . LOOP "
324 I$=I$+B$
326 K=K+1
328 GOTO 70
330 REM
902 IF A$<>"+" THEN GOTO 910
904 N=N-1
906 S(N)=S(N)+S(N+1)
908 GOTO 100
910 IF A$<>"-" THEN GOTO 920
912 N=N-1
914 S(N)=S(N)-S(N+1)
916 GOTO 100
920 IF A$<>"*" THEN GOTO 930
922 N=N-1
924 S(N)=S(N)*S(N+1)
926 GOTO 100
930 IF A$<>"/" THEN GOTO 940
932 N=N-1
934 S(N)=S(N)/S(N+1)
936 GOTO 100
940 IF A$<>"ABS" THEN GOTO 950
942 S(N)=ABS(S(N))
944 GOTO 100
950 IF A$<>"ATN" THEN GOTO 960
952 S(N)=ATN(S(N))
954 GOTO 100
960 IF A$<>"COS" THEN GOTO 970
962 S(N)=COS(S(N))
964 GOTO 100
970 IF A$<>"EXP" THEN GOTO 980
972 S(N)=EXP(S(N))
974 GOTO 100
980 IF A$<>"INT" THEN GOTO 990
982 S(N)=INT(S(N))
984 GOTO 100
990 IF A$<>"LOG" THEN GOTO 1000
992 S(N)=LOG(S(N))
994 GOTO 100
1000 IF A$<>"RND" THEN GOTO 1010
1002 S(N)=RND(-N)
1004 GOTO 100
1010 IF A$<>"SGN" THEN GOTO 1020
1012 S(N)=SGN(S(N))
1014 GOTO 100
1020 IF A$<>"SIN" THEN GOTO 1030
1022 S(N)=SIN(S(N))
1024 GOTO 100
1030 IF A$<>"SQR" THEN GOTO 1040
1032 S(N)=SQR(S(N))
```

Listing. Ein Forth-Interpreter zum Abtippen

```
1034 GOTO 100
1040 IF A$<>"TAN" THEN GOTO 1050
1042 S(N)=TAN(S(N))
1044 GOTO 100
1050 IF A$<>"^" THEN GOTO 1060
1052 S(N)=S(N)^S(N+1)
1054 GOTO 100
1060 IF A$<>"S?" THEN GOTO 1070
1062 FOR I=1 TO N
1064 PRINT S(N-I+1)
1066 NEXT I
1068 GOTO 100
1070 IF A$<>"." THEN GOTO 1080
1071 IF N<1 THEN GOTO 106
1072 PRINT S(N)
1074 N=N-1
1076 GOTO 100
1080 IF A$<>"DUP" THEN GOTO 1090
1082 N=N+1
1084 S(N)=S(N-1)
1086 GOTO 100
1090 IF A$<>"DROP" THEN GOTO 1100
1092 N=N-1
1094 GOTO 100
1100 IF A$<>"SWAP" THEN GOTO 1110
1102 S(N+1)=S(N-1)
1104 S(N-1)=S(N)
1106 S(N)=S(N+1)
1108 GOTO 100
1110 IF A$<>"OVER" THEN GOTO 1120
1112 N=N+1
1114 S(N)=S(N-2)
1116 GOTO 100
1120 IF A$<>">R" THEN GOTO 1130
1122 M=M+1
1124 R(M)=S(N)
1126 N=N-1
1128 GOTO 100
1130 IF A$<>"R>" THEN GOTO 1140
1132 N=N+1
1134 S(N)=R(M)
1136 M=M-1
1138 GOTO 100
1140 IF A$<>"R@" THEN GOTO 1200
1142 N=N+1
1144 S(N)=R(M)
1146 GOTO 100
1200 REM
1202 IF A$<>"=" THEN GOTO 1210
1203 N=N-1
1204 IF S(N)=S(N+1) THEN GOTO 1207
1205 S(N)=0
1206 GOTO 100
1207 S(N)=1
1209 GOTO 100
1210 IF A$<>">" THEN GOTO 1220
1212 N=N-1
1214 IF S(N)>S(N+1) THEN GOTO 1217
1215 S(N)=0
1216 GOTO 100
1217 S(N)=1
1218 GOTO 100
1220 IF A$<>"<" THEN GOTO 1230
1222 N=N-1
1223 IF S(N)<S(N+1) THEN GOTO 1227
1224 S(N)=0
1225 GOTO 100
1227 S(N)=1
1228 GOTO 100
1230 IF A$<>"IF" THEN GOTO 1250
1231 N=N-1
1232 IF S(N+1) THEN GOTO 100
1233 FOR I=L(K) TO LO(K)-3
1234 B$=I$(I,I+3)
1235 IF B$="ELSE" THEN GOTO 1240
1236 IF B$="THEN" THEN GOTO 1240
1237 NEXT I
1238 PRINT "IF?"
1239 GOTO 30
1240 L(K)=I+4
1241 GOTO 100
1242 GOTO 100
1250 IF A$<>"ELSE" THEN GOTO 1260
1252 GOTO 1233
1260 IF A$<>"THEN" THEN GOTO 1270
1262 GOTO 100
1270 IF A$<>"BEGIN" THEN GOTO 1280
1272 M=M+1
1274 R(M)=L(K)
1276 GOTO 100
1280 IF A$<>"UNTIL" THEN GOTO 1300
1282 N=N-1
1283 IF S(N+1) THEN GOTO 1288
1284 IF S(N+1) THEN GOTO 100
1286 L(K)=R(M)
1287 GOTO 100
1288 M=M-1
1289 GOTO 100
1300 IF A$<>"DO" THEN GOTO 1320
1302 M=M+1
1304 R(M)=L(K)
1305 M=M+1
1306 R(M)=S(N-1)
1308 M=M+1
1309 R(M)=S(N)
1310 N=N-2
1312 GOTO 100
1320 IF A$<>"LOOP" THEN GOTO 1340
1322 R(M)=R(M)+1
1324 IF R(M-1)>R(M) THEN GOTO 1330
1326 M=M-3
1328 GOTO 100
1330 L(K)=R(M-2)
1332 GOTO 100
1340 REM
1500 IF A$<>"PI" THEN GOTO 1510
1502 N=N+1
1504 S(N)=3.14159
1506 GOTO 100
1510 IF A$<>"0" THEN GOTO 1520
1512 N=N+1
1514 S(N)=0
1516 GOTO 100
1520 IF A$<>"STOP" THEN GOTO 1600
1522 STOP
1600 NUM=1
1602 FOR I=1 TO LEN(A$)
1604 IF MID$(A$,I,1)<"0" OR MID$(A$,I,1)
>"9" THEN NUM=0
1606 IF I=1 AND MID$(A$,1,1)="-" THEN NU
M=1
1608 NEXT I: IF NUM=0 THEN PRINT A$;"  N
OT DEFINED": GOTO 30
1610 N=N+1
1612 S(N)=VAL(A$)
1614 GOTO 100
```

Listing. Ein Forth-Interpreter zum Abtippen (Schluß)

Trace-Befehl für FIG-Forth

Hier wird ein neues, nützliches Forth-Wort vorgestellt, mit dem man den Ablauf anderer Wörter schrittweise untersuchen kann.

Die einfachste Art, ein Forth-Wort zu testen, besteht darin, es mit den entsprechenden Eingangswerten auf dem Stack aufzurufen und dann zu hoffen, daß sich kein Fehler eingeschlichen hat. Wenn sich der Computer dann noch normal zurückmeldet, der richtige Wert ausgegeben wird und der Stack sich in dem Zustand befindet, in dem er sich auch befinden sollte, so kann angenommen werden, daß das Wort richtig arbeitet. Manchmal aber — und das kommt öfter vor, als man möchte — verliert sich der Computer in einem Irrgarten und nichts läuft mehr. In diesem Falle ist es sehr nützlich, ein Wort zur Verfügung zu haben, das etwa einem TRACE-Befehl in Basic entspricht. Solch ein TRACE ist aber nicht ganz einfach zu realisieren. Der Grund dafür ist unter anderem darin zu suchen, daß es in Forth nicht nur Wörter gibt, die nur 2 Byte Platz in dem Parameterfeld beanspruchen, sondern auch ein paar andere, die neben ihrer CFA (Code-Feld-Adresse) auch noch Daten ablegen. Dazu gehören alle strukturierenden Wörter (IF, ELSE, THEN / DO, LOOP, +LOOP / BEGIN, UNTIL, WHILE, REPEAT, AGAIN), die entweder BRANCH oder auch 0BRANCH compilieren und daran noch ihre Sprungweite anhängen. Ebenso zählen LIT und CLIT dazu, die außer ihrer CFA noch den Wert der Konstanten eintragen. Nicht zu vergessen auch das Wort »."«, das ganze Texte in das Parameterfeld speichert.

Alle Wörter, die in irgendeiner Weise den Returnstack manipulieren, sind mit besonderer Vorsicht zu behandeln, da ein TRACE-Wort diesen Stack selber benötigt. Die Wörter R⟩, R, DO, LOOP, +LOOP, I, I', J, K, LEAVE gehören dazu und müssen deshalb von TRACE alle gesondert behandelt werden. Man muß für jedes dieser Wörter eine Routine schreiben, die an einem eigenen Stack diese Manipulationen entsprechend dem originalen Programm durchführt. Alle angegebenen Wörter müssen von »TRACE« gesondert behandelt werden. Der ganze Rest aber kann von dem internen Interpreter ausgeführt werden. Für sämtliche Returnstack-Manipulationen benötigt man neben einem eigenen Returnstack auch noch den entsprechenden Pointer.

ONE-STEP ist kein neuer Tanzschritt

Am einfachsten erscheint es deshalb, TRACE als Rahmenprogramm aufzufassen, das die Ein- und Ausgaben durchführt und mit dem Anwender kommuniziert. Muß dann einmal ein Wort ausgeführt werden, wird ONE-STEP (Listing) aufgerufen, das dann das Wort ausführt, auf das der selbst definierte Instruction-Pointer zeigt.

In dem Wort ONE-STEP sind dann sämtliche Fälle, die nicht von dem inneren Interpreter ausgeführt werden können, einzeln abzuarbeiten.

Mit dem Wort ONE-STEP können fast alle Forth-Wörter getestet werden. Falls es sich bei dem Wort um ein Primitive, eine Konstante oder Variable handeln sollte, so wird Ihnen das sofort mitgeteilt. Nur Forth-Wörter, die in Highlevel verfaßt sind, werden von TRACE auch entsprechend behandelt. Wenden Sie TRACE auch mal auf Wörter des Kernals an. Dadurch erhalten Sie einen guten Einblick in die Arbeitsweise von TRACE, und Sie lernen so auch sehr gut die Programmierung in Forth selbst kennen.

Hier ein paar Beispiele:

```
4 TRACE ·
: · S-⟩D D.4 ;
4 . TRACE D.
: D. 0 D.R SPACE 4 ;
4 . 0 TRACE D.R
: D.R ⟩R SWAP OVER DABS ⟨#   #S SIGN #⟩ R⟩ OVER
SPACES TYPE 4 ;
```

Sie sehen daran, wie eng verknüpft Forth selbst in so einem grundlegenen Wort wie ».« ist. Auch die Decompilereigenschaften von Forth sind hier ein wenig dargestellt.

TRACE funktioniert so lange als Decompiler, bis ein Wort auszuführen ist, das in seinem Verlauf eine Ausgabe durchführt. Diese Aufgabe, die normalerweise als einzige auf dem Bildschirm erscheinen würde, steckt jetzt mitten in dem entschlüsselten Wort und macht es so manchmal etwas problematisch, den genauen Sourcetext zu erkennen. Weiterhin werden alle strukturierenden Wörter nicht angezeigt, sondern nur deren »Run Time Executive« (BRANCH oder 0BRANCH) und eventuell auch ausgeführt.

Es kann sein, daß die Version Ihres Forth nicht ganz genau mit der übereinstimmt, die wir eingesetzt haben. So ist zum Beispiel das Wort CLIT nicht in jeder Version implementiert. Es kann sein, daß Sie das Wort DLIT in Ihrem Programm vertreten haben.

Einfache Anpassung

Für den Fall, daß Sie kein CLIT haben, lassen Sie die ganze Zeile einfach unter den Tisch fallen und tippen gleich ein THEN weniger ein.

Ist das Wort DLIT aber in Ihrer Version enthalten, so müßten Sie einfach eine neue Zeile einfügen, ganz der Zeile von CLIT entsprechend. Der Unterschied zu CLIT besteht in der Änderung von: 1. Lesebefehl C@ nach D@ (Liest statt einem Byte ein Langwort, 32 Bit), 2. IPOI darf nicht nur um 1 erhöht, sondern muß um 4 inkrementiert werden.

In dem Falle, daß sonst noch irgendwelche Wörter in Ihrem Forth vorhanden sind, die Daten mit in das Parameterfeld mit ablegen (dies kommt oft bei Erweiterungen vor, Strings, spezielle Datenwörter), so müßten Sie diese Wörter selber behandeln. Entsprechendes gilt auch für die Returnstackmanipulierenden Wörter.

Nicht anwenden sollten Sie TRACE bei Wörtern, die selbst nur zur Definition von anderen Datentypen entwickelt wurden. Insbesondere die ⟨BUILDS .... DOES⟩-Funktion funktioniert nicht ganz vollständig mit unserem »Trace«. Auch mit anderen Wörtern mag es Schwierigkeiten geben, doch diese Wörter befinden sich in der Unterzahl.

Noch ein letzter Hinweis. Der Stackpointer muß nach ordnungsgemäßer Beendigung des Wortes auf 0 stehen. Jeder andere Wert in SPOI würde bei normalem Ablauf durch den inneren Interpreter zum Absturz des Systems führen. Denken Sie bitte auch daran: Wenn das Wort durch »;« beendet wird, muß in SPOI eine Null stehen. Deshalb lasse ich den Wert dieser Variablen beim Abbruch auch gleich mit ausdrucken.

Für etwaige Anregungen oder Verbesserungsvorschläge sind wir dankbar.

(Bernhard Leikauf/ev)

```
0 VARIABLE IPOI                           (Def. Instruction Pointer)
0 VARIABLE SPOI                           (Def. Returnstack Pointer)
0 VARIABLE RSTA 38 ALLOT                  (Def. Returnstack für max. 20 Einträge)
: ONE-STEP IPOI @                         (Hole IP aus IPOI)
DUP @ ' OBRANCH CFA =                     (Wort = OBRANCH)
 IF SWAP IF DROP 2 ELSE                   (Nimm Flag vom Stack)
2+ @ THEN IPOI +! ELSE                    (True Flag. Dann überspringe den Offset False Flag. Addiere den Offset zu IPOI;)
                                          (führe den Sprung aus)
DUP @ ' BRANCH CFA =                      (Wort = BRANCH)
 IF  2+ @ IPOI +! ELSE                    (Addiere den Offset zu IPOI; führe den Sprung immer durch)
DUP @ ' LIT CFA =                         (Wort = LIT+)
 IF 2+ @ DUP . 2 IPOI +! ELSE             (Hole den 16-Bit-Wert, zeige ihn an und lege ihn auch auf dem Stack ab. Setze IPOI auf das)
                                          (Wort nach dieser Zahl)
DUP @ 'CLIT CFA =                         (Wort = CLIT)
IF 2+ C@ DUP . 1 IPOI +! ELSE             (Hole den 8-Bit-Wert, zeige ihn an und lege ihn auch auf dem Stack ab. Setze IPOI auf das)
                                          (Wort nach dieser Zahl)
DUP @ ' (.'') CFA =                       (Wort = (.''))
 IF 2+ COUNT DUP 1+ IPOI +! TYPE 32 EMIT  (Versetze IPOI auf das Wort hinter dem Text und drucke diesen Text mit »Space« aus)
ELSE
DUP @ ' )R  CFA =                         (Wort = )R)
 IF DROP RSTA SPOI @+ !                   (Speichere den Wert auf dem Stack an die)
2 SPOI+! ELSE                             (Adresse, auf die der Stackpointer zeigt und erhöhe dann den Stackpointer um 2)
DUP @ ' R  CFA = OVER @ ' I CFA = OR      (Wort = R oder = I  ? dasselbe PRG)
IF DROP RSTA SPOI @+ 2- @ ELSE            (Kopiere den Wert, auf den der Stackpointer zeigt, auf den Parameterstack)
DUP @ ' R)  CFA =                         (Wort = R) )
 IF DROP RSTA SPOI @+ 2- @-2 SPOI +!      (Hole den obersten Eintrag des Returnstacks auf den P.stack und dekrementiere den)
ELSE                                      (Stackpointer SPOI um 2)
DUP @ ' (DO)  CFA = IF DROP SWAP RSTA     (Speichere den Schleifenindex und das)
SPOI @+ 2! 4 SPOI +! ELSE                 (Maximum auf dem eigenen R.stack ab)
DUP @ ' I'  CFA =                         (Wort = I')
 IF DROP RSTA SPOI @ + 4 - @ELSE          (Hole den zweitobersten Eintrag auf den P.stack)
DUP @ 'J  CFA =                           (Wort = J)
 IF DROP RSTA SPOI @ + 6 - @ELSE          (Hole den dritten Eintrag des R.stacks)
DUP @ 'K  CFA =                           (Wort = K)
 IF DROP RSTA SPOI @ + 10 - @ELSE         (Hole den fünften Eintrag des R.stacks)
DUP @ ' LEAVE  CFA =                      (Wort = LEAVE)
 IF DROP RSTA SPOI @ + 2 - DUP @ SWAP 2-  (Ändere das Max. der Schleife auf den gegenwärtigen Wert des Schleifenindex (I) ab)
! ELSE
DUP @ ' (LOOP)  CFA =                     (Wort) = (LOOP)
 IF RSTA SPOI @ + 2- DUP 1+! DUP 2- @     (Inkrementiere den Schleifenindex. Hat er das Max. erreicht?)
SWAP @)
 IF 2+ @ IPOI +!                          (Führe Sprung zum Schleifenstart aus, wenn I' ) I)
ELSE DROP -4 SPOI +! 2 IPOI +!            (Schleifenende erreicht. Verringere Stackpointer)
THEN ELSE                                 (um 4 (Index und Max.. Dann überspringe den Offset zum Schleifenanfang)
DUP @ ' (+LOOP) CFA =                     (Wort = (+LOOP))
 IF RSTA SPOI @ + 2- ROT OVER +!DUP 2- @  (Erhöhe den Schleifenparameter um den angegebenen Wert)
SWAP @)                                   (Ist jetzt das Maximum erreicht oder schon überschritten?)
 IF 2+ @ IPOI +!                          (Selbe Operation wie bei (LOOP))
ELSE DROP -4 SPOI +! 2 IPOI +!
THEN  ELSE                                (Eventuelle weitere Abweichungen von der Norm wären dann hier fortlaufend einzutragen)
@ EXECUTE                                 (Andernfalls ist es ein vom inneren Interpreter)
                                          (ausführbares Wort und so auch von diesem zu erledigen)

THEN THEN THEN THEN THEN THEN THEN THEN
THEN THEN THEN THEN THEN THEN THEN
2 IPOI +!                                 (Abschluß sämtlicher eröffneter Abfragen, stelle nach Erledigen des Wortes IPOI auf)
                                          (das nächste zu erledigende Wort).
```

```
: TRACE  -Find
   IF DROP CFA DUP @ ' ONE-STEP CFA @ =
     IF CR .'' : '' 2+ DUP NFA ID. IPOI ! 0 SPOI !
       BEGIN IPOI @ @ DUP 2+ NFA ID.
             BEGIN (Warteschleife des Rechners) UNTIL
             ' ;S CFA = ?TERMINAL OR IF CR SPOI ? [COMPILE],S THEN
             ONE-STEP
        AGAIN
     ELSE .'' NO HIGH LEVEL '' DROP THEN
   ELSE .'' NOT FOUND '' THEN ;
```

Listing. Das vollständige »Trace«-Programm. Die Kommentare sind nicht einzugeben.

Turtle-Grafik mit Forth

Als Beispiel für ein komplexes Programm in Forth stellen wir hier ein Grafik-Paket vor.

Obwohl ursprünglich für den C64 mit HES-Forth geschrieben, lohnt es sich sicher auch für die Besitzer anderer Computer oder anderer Forth-Versionen, sich mit diesem Grafikpaket etwas näher zu beschäftigen. HES-Forth ist im wesentlichen ein etwas erweitertes FIG-Forth. Zur Anpassung an andere Computer brauchen nur die systemspezifischen Teile dieses Programm-Pakets geändert werden.

Im folgenden Text geben wir eine ausführliche Programmbeschreibung, die die Arbeit mit diesem Programm erleichtern soll. Das gesamte Paket besteht aus drei Teilen:
- dem High-Resolution-Graphic-Package
- der Multi-Color-Graphic
- den Extras

Das Programm wird durch die Lade-Screens 02, 03 und 04 des Main-File geladen.

Es steht eine hochauflösende Grafikseite mit einer Auflösung von 320 x 200 Pixel zu Verfügung. Der Punkt (0,0) liegt dabei in der linken oberen Ecke. Die Befehle HION und HIOFF schalten zwischen dem normalen Arbeitsbildschirm und der Grafikseite hin und her. Alle Eingaben können weiterhin im direkten Modus erfolgen. Ein Beispiel sieht so aus:

```
HION              ( Einschalten der Grafik )
14 0 CFILL        ( Farben einstellen )
HCLEAR            ( Bildschirm löschen )
0 0 319 199 LINK  ( Diagonale ziehen )
HIOFF             ( Ausschalten der Grafik )
```

Entsprechend der üblichen Umgekehrt Polnischen Notation erwarten alle Worte ihre Parameter auf dem Stack. Wichtig sind dabei immer die Leerzeichen (Spaces) zwischen den Wörtern. Solange die Grafikseite aktiv ist, sind die Eingaben ja nicht sichtbar.

Bei einem Tippfehler drückt man entweder RUN/RESTORE - dabei wird der Stack und der Arbeitsbildschirm gelöscht, nicht aber die Grafikseite - und wiederholt die letzte Eingabe. Oder aber man gelangt durch HIOFF zurück auf den Arbeitsbildschirm, der die letzten Eingaben zeigt. Der Handler zur Adreßberechnung für X-Werte außerhalb (0/319) und Y-Werte außerhalb (0/199) steht in XFORM beziehungsweise YFORM und kann vom Benutzer abgeändert werden. XFORM und YFORM sind als »wrap-around« initialisiert, das heißt (-1,-1) ist gleich (319,319) etc.

Die Grafik kann als sequentielle Datei auf Diskette gespeichert werden, und zwar mittels der Befehle HIWRITE und HIREAD. Zuvor ist einmal mit »SEQ name« ein Filename festzulegen. Die beiden Befehle beziehen sich so lange auf name, bis dieser mit SEQ geändert wird.

Der Aufbau der Datei sieht folgendermaßen aus:

8 KByte	Bit-Maß
1 Byte	Hintergrundfarbe
1000 Byte	Farb-RAM (low)
1000 Byte	Farb-RAM (high)

PLOT und LINK stehen sowohl als High-Level-Wort wie auch als Primitive zur Verfügung. Welche Version geladen werden soll, wird mit Scr #02 eingestellt.

Nun zum Laden:

```
FILE MAIN
2 LOAD
FCLOSE
```

Der Ladevorgang nimmt etwa vier Minuten in Anspruch.

Zusätzlich zur normalen hochauflösenden Grafik bietet das Paket eine Multi-Color-Grafik mit 160 x 200 doppelt-breiten Punkten, wobei jedes Pixel eine von vier möglichen Farben haben kann. Das Ein-/Ausschalten nehmen MON (Multi-Color ein) und MOFF (aus) vor. CFILL erwartet jetzt vier Parameter (Farben) auf dem Stack. HCOL wählt die Zeichenfrabe 1,2 oder 3 aus dem mit MCOL definierten Farb-Set.

Da alle Tastatur-Eingaben das Farb-RAM (high) beeinflussen, sollte man im direkten Modus nicht mit mehr als vier Farben (ändern mit MCOL) arbeiten. ?LOAD veranlaßt beim Laden der »Extras« das zusätzliche Einlesen von MPUT und MSHAPER.

Die übrigen Befehle entsprechen den normalen Hires-Befehlen. Das Laden der Multi-Color-Grafik erfolgt mit

```
FILE MAIN
3 LOAD
FCLOSE
```

und beansprucht etwa vier Minuten.

Neben diesen beiden grundsätzlichen Einstellungen findet man zusätzlich noch folgende Extras.

```
— PAINTER    (Zeichenprogramm)
— SHAPES     (der SHAPER erfordert den PAINTER)
— STRINGS    (benötigen die SHAPES)
— TURTLE     (Turtle Grafik
```

Painter

Per Tastatur-Steuerung kann ein einzelner Punkt auf dem Bildschirm bewegt werden und Linien zeichnen oder löschen. Der Aufruf erfolgt mit »x y PAINT«, worauf der »Zeichenstift« bei (x,y) erscheint. RETURN beendet den Zeichenvorgang, aber die letzten Zeichenkoordinaten verbleiben auf dem Stack. So kann man entweder mit UNPLOT den letzten Zeichenpunkt löschen oder irgendwelche Zwischenrechnungen, Farbänderungen etc. vornehmen und den PAINTER erneut mit PAINT aufrufen. Es wird immer an der letzen Stelle weitergezeichnet.

Zur Steuerung folgende Details:
- Farbeinstellung: Funktionstasten f1, f3, f5, f7
- Pen-Up/Pen-Down: G
- Bewegen des Zeichenpunktes in acht Himmelsrichtungen: R,T,Y,F,H,V,B,N

Shapes

Ein Shape ist eine x mal y Punktmatrix, wobei x ein Vielfaches von 8 ausmacht. Ein 3 x 21-Shape besteht demnach aus 24 x 21 Punkten, hat also die Größe eines Sprites. SHAPE ist eine Compiler-Erweiterung, die beliebig dimensionierte Shapes erzeugt. Der Aufruf »x y SHAPE name« definiert ein x mal y-SHAPE genannt »name«. Die Eingabe dieses Namens bewirkt dann stets den Aufruf.

```
3 21 SHAPE PETRA     (definiert PETRA)
PETRA ?S             (zeigt PETRAs Datas an)
PETRA CLEAR          (löscht PETRA)
PETRA ?S
HION
PETRA SHAPER         (zeichnet Rahmen um PETRA und
                      aktiviert den Painter)
PETRA 100 100 PUT    (zeichnet PETRA)
PETRA 110 110 PUT
HIOFF
```

Anstelle von »name« kann auch »n SPRITE« stehen, wobei n ($0 \leq n \leq 7$) sich auf das Sprite mit der Nummer n bezieht. Zuvor aber sollten alle Sprite-Pointer mit !POINTER einmal gesetzt worden sein.

Die Shapes (beziehungsweise Sprites) lassen sich mit

»name SHAPEWRITE« unter dem mit SEQ definierten Namen als sequentielles File ablegen. Ein Beispiel:

```
SEQ GIRL
PETRA SHAPEWRITE    (legt ein sequentielles File
                     namens GIRL an, dessen
                     Punktemuster PETRA entspricht)
```

Wird im Multi-Color-Modus gearbeitet, sollten Sie möglichst MPUT anstatt PUT und MSHAPER statt SHAPER verwenden.

Strings

Diese Anweisung stellt ein kleines String-Paket dar und zählt ebenfalls zu den Compiler-Erweiterungen. Strings sind wie Variable vor Benutzung zu definieren, also »name«. Ihnen stehen dann zwei Eingabevarianten frei: »name INPUT« entspricht in Basic »GETA« oder »name =$ text«, das »A$="text"« gleichkommt. Die Ausgabe erfolgt mit »name« (Arbeitsbildschirm) oder »name x y PUT$« (Grafikseite).

»n CHR« verhält sich wie ein 1 x 8-Shape, dessen Punktemuster dem Zeichen mit dem Bildschirm-Code n entspricht.

Turtle

Die Turtle-Grafik baut auf dem Hires- und Multi-Color-Paket auf und umfaßt alle Grafikbefehle des Commodore-Logo. Es beansprucht lediglich 456 Byte (!), was die Leistungsfähigkeit von Forth gut illustriert. Die Koordinaten der Turtle stehen immer als oberste Zahlen auf dem Stack (Achtung!), also die Richtung der Schildkröte in der Variablen HEADING. Bei der Übertragung von Logo-Programmen muß die UPN von Forth beachtet werden. Statt »FORWARD 10« erwartet Forth die 10 auf dem Stack, also »10 FORWARD«.

Der Punkt (0,0) liegt wie gewohnt in der linken oberen Ecke und nicht, wie bei den meisten Logo-Versionen, in der Bildschirmmitte. Dies kann der Benutzer durch »: MOVE LFORN...;« und entsprechende Definition von LFORN nach Bedarf ändern.

Alle Turtle-Befehle können natürlich wie bei Logo abgekürzt werden. Sie schreiben dann statt »10 FORWARD« einfach »10 FD« und so fort. Die Turtle sollte sich nicht mehr als 30000 Punkte in jeder Richtung vorwärts bewegen (2-Byte-Arithmetik). Der Befehl TURTLE initialisiert die Schildkröte und das Farb-RAM, löscht die Grafikseite und sollte nur dann aufgerufen werden, wenn man sich das HION ersparen will – also nicht von der Grafikseite aus. Ein Beispielprogramm:

```
: LINIE 50 FD 90 RT ;
: QUADRAT LINIE LINIE LINIE LINIE
: ROSETTE 36 0 DO QUADRAT 10 RT 12 FD LOOP ;

TURTLE ROSETTE
```

Soweit der allgemeine Überblick über die einzelnen, im Turtle-Forth enthaltenen Programm-Pakete.

Die Screens von Turtle-Forth

SCR # 1: nicht verwendet.

SCR # 2 bis 4: Lade-Screens. SCR # 2 lädt die normale Hires-Grafik, wobei die Befehle PLOT und LINK wahlweise als Primitive (SCR # 70 bis 77) oder als High-Level (SCR # 12 bis 13 und 20) definiert werden – je nachdem wie die Klammern in SCR # 2 stehen.

SCR # 3 lädt die Multi-Color-Grafik, bei der sowohl der normale als auch der Multi-Color-Modus zur Auswahl stehen kann. Die Umschaltung zwischen den beiden Modi erfolgt Mit den Wörtern MON beziehungsweise MOFF. Zu beachten ist, daß der Dictionary-Pointer zwischendurch auf 16384 hochgesetzt wird, um 8 KByte Platz für die Bitmap zu schaffen (siehe rechts). SCR # 4 lädt die Extras, die natürlich auch einzeln zu verwenden sind.

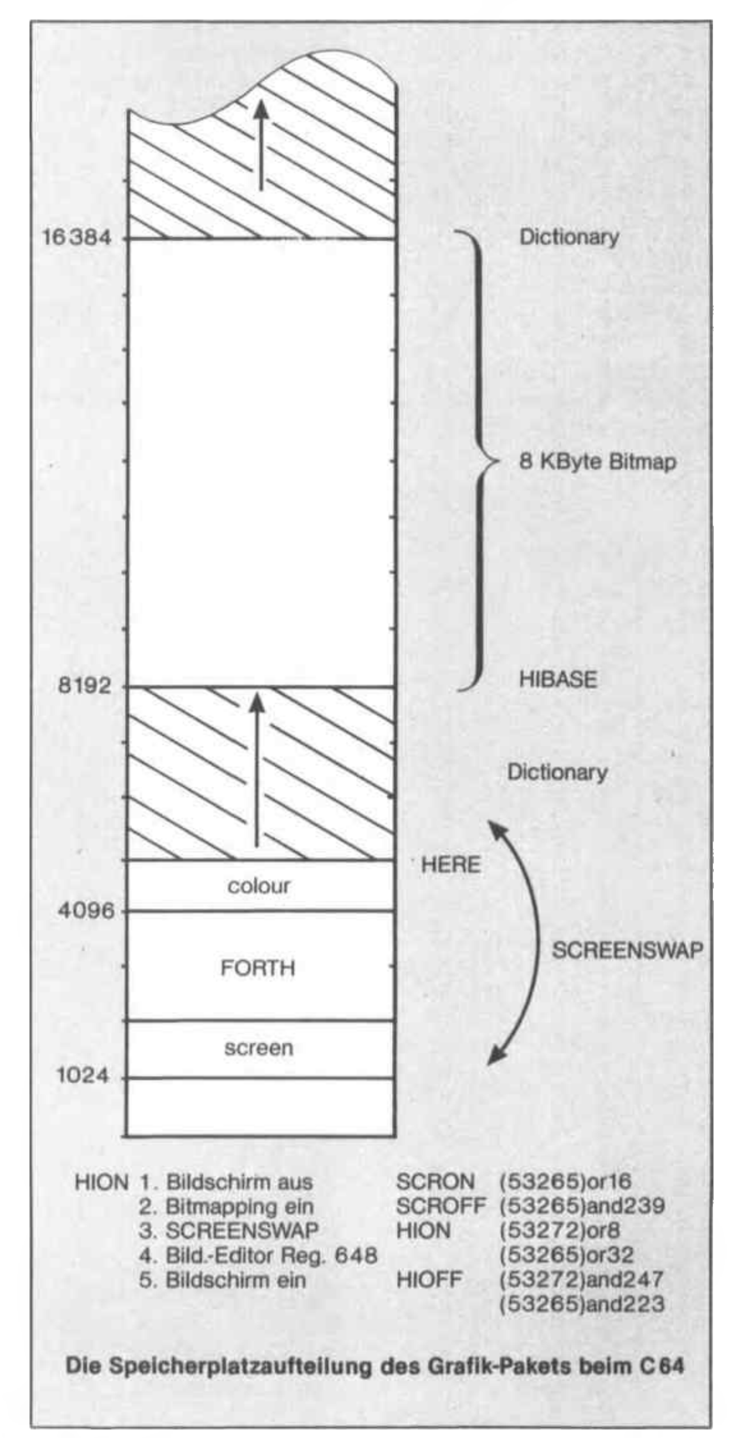
SCR # 5 bis 9: nicht verwendet.

SCR # 10 bis 11: Diese beiden Screens enthalten die Installation der hochauflösenden Grafik und müssen dem Computertyp angepaßt werden. Die hier vorgestellten Wörter gelten für den Commodore 64.

Zunächst werden 1000 Byte Platz geschaffen, um den Bildschirm zwischenzuspeichern. Das bezweckt, daß Sie auch bei eingeschalteter Hires-Grafik weiterhin alle Befehle im Direktmodus eingeben können. SCREENSWAP besorgt die Speicherverschiebung und benutzt das Wort NSWAP (adr 1 adr 2 n –), welches n Byte zwischen den Adressen adr_1 und adr_2 austauscht. SCRON beziehungsweise SCROFF schalten den Videochip ein und aus, um einen sauberen Übergang beim Einschalten der Grafik zu erzielen.

HION und HIOFF besorgen die Umschaltung auf hochauflösende Grafik. Falls Sie mit einem anderen Computer als dem C 64 arbeiten, müssen hier natürlich andere »Pokes« stehen, die Sie dem Benutzerhandbuch entnehmen können.

Falls die Hires eingeschaltet ist, legt HION? ein Flag (ungleich Null) auf den Stack, ansonsten eine Null. SCREEN teilt dem Bildschirmeditor des Betriebssystems mit, wo der



Die Speicherplatzaufteilung des Grafik-Pakets beim C64

Eingabe-Bildschirm liegt. Bei ausgeschalteter Hires befindet er sich ab Register 1024 (also Page 4), bei eingeschalteter Hires, also nach SCREENSWAP ab Register 4096 (also Page 16).

Bei Forth-Programmen empfiehlt es sich, Wort für Wort einzugeben und auszutesten. Dann weiß man nämlich sicher, wo man steht und spart sich viel Ärger beim Fehlersuchen. Sie kennen nun alle Wörter, um mit der hochauflösenden Grafik auch umzugehen.

SCR # 12 bis 13: Mit PLOT und UNPLOT setzen oder löschen Sie die einzelnen Punkte. -PLOT bewirkt PLOT oder UNPLOT je nach Verhältnis der Variablen VPLOT und VBREAK. Es wird von LINK benutzt. Dadurch ist auch UNLINK einfach als Spezialfall von LINK zu definieren und zudem lassen sich alle Linien auch unterbrochen (BREAK) darstellen.

ADR bewirkt die Adreß-Umrechnung von X/Y-Koordinaten in die Registeradresse der Bitmap.

SCR # 14 bis 15: Bevor wir mit der Hires (CIRCLE) fortfahren, wollen wir die Multi-Color-Grafik installieren. SCR # 14 und 15 entsprechen bis auf kleine Details SCR # 10 und 11.

SCR # 16 bis 19: Hier wirkt eine trickreiche Programmier-Technik. Damit sowohl die Arbeit mit normaler Hires als auch mit Multi-Color-Hires möglich ist, sind die meisten Wörter zweimal zu definieren. Um den gleichen Namen wieder zu verwenden, werden die Wörter vektoriell ausgeführt.

Zum Beispiel PLOT. Es führt das Wort aus (EXECUTE), das in der Variablen 'PLOT steht. Dort befindet sich im normalen Hires-Modus die Code-Feld-Adresse (CFA) von PLOT,H im Multi-Color-Modus dagegen die CFA von PLOT,M. Das Einbeziehungsweise Ausschalten mit MON und MOFF ändert also lediglich die Vektoren 'PLOT 'ADR etc. ab. MCON und MCOFF bewirken dabei die eigentlichen »Pokes«, um den entsprechende Grafik-Modus einzuschalten.

SCR # 20: Auf dem PLOT-Befehl aufbauend zeichnen wir jetzt eine Gerade, das heißt, alle Punkte zwischen zwei Koordinaten. LINK verwendet, je nachdem, ob die Steigung der Geraden mehr als 45 Grad beträgt, LINX oder LINY. Nun gibt auch BREAK einen einleuchtenden Sinn.

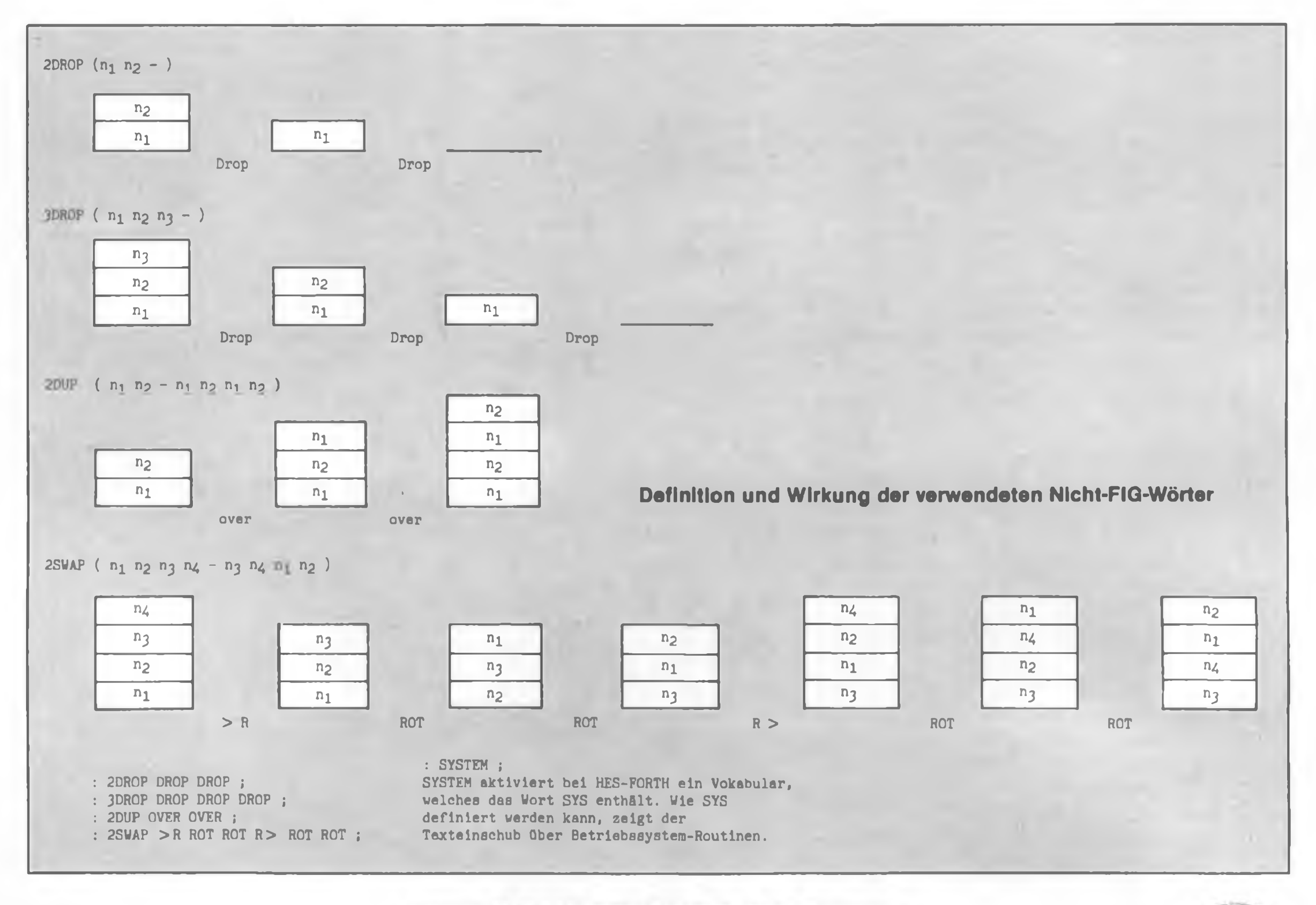
SCR # 21 bis 23: Um einen Kreis zu zeichnen, benötigen wir eine Sinus- oder Cosinus-Funktion. Das scheint schwierig, da Forth doch keine Fließkommazahlen kennt.

Wir erzeugen uns jedoch einfach Grad für Grad eine Liste der Sinuswerte. (Wegen der Beziehung $\cos \alpha = \sin (90-\alpha)$ kann man sich eine Cosinus-Liste sparen.) Das entscheidende Wort ist nun CIRC. Es berechnet aus dem Winkel a die Koordinaten x und y unter Benutzung des Radius, der in der Variablen RAD stehen soll. ARC zeichnet dann einen Kreisbogen, wobei die mit CIRC gewonnenen Koordinaten mit dem Quotienten EPS/100 multipliziert werden, so daß sich die Kreise stauchen und dehnen lassen. EPS ist zu 100 initialisiert. Es entstehen also wirklich Kreise, solange man die Exzentrität nicht mit EXCENTER ändert. CIRCLE ist dann nur noch ein Spezialfall von ARC.

SCR # 24: SCR # 24 demonstriert schön die Forth-Philosophie: NGON zeichnet offene, POLYGON geschlossene Kurvenzüge. Beide benutzen dazu ein Wort namens GON. TRIC und REC, also Drei- und Vierecke – Spezialfälle geschlossener Kurven – arbeiten also mit POLYGON. QUADRAT wiederum ist ein Sonderfall von REC.

SCR # 25 bis 27: Hier wird das Wort SYS (a x y adr – a x y) benötigt, das nicht dem FIG-Wortschatz entstammt. Die FIG-Definition eines solchen Wortes findet sich im Texteinschub »Einbinden von Betriebssystem-Routinen«. SF ist eine Stringvariable, die mit SEQ eingelesen und mit .SEQ ausgegeben wird.

SWRITE ruft einige Betriebssystem-Routinen zur sequentiellen Datenspeicherung auf:



Definition und Wirkung der verwendeten Nicht-FIG-Wörter

```
$ FFBD   Setnam
$ FFBA   Setlfs
$ FFC0   Open
$ FFC9   CHKOUT
```

SREAD ruft auf:

```
$ FFBD   Setnam
$ FFBA   Setlfs
$ FFC0   Open
$ FFC6   CHKIN
```

SCLOSE ruft auf:

```
$ FFC3   Close
$ FFCC   Clrchn
```

So lassen sich einzelne Bytes (TO FROM) und ganze Speicherbereiche (TOS FROMS) auf Diskette sequentiell speichern. Diese Routinen sind Commodore-spezifisch und sind auf anderen Computern unbrauchbar. Die Befehle sind aber sicherlich für viele C 64-Forth-Programme äußerst nützlich.

SCR # 28: HIWRITE und HIREAD benutzen nun die Fähigkeit zu sequentieller Datenspeicherung, um Grafiken zu speichern und wiederzulesen. Folgende Adreßbereiche können bearbeitet werden.:

1. 8 KByte Bitmap (8192 bis 16383)
2. Hintergrund (53281)
3. Bildschirm (1024 bis 2023 beziehungsweise 4096 bis 5095)
4. Farbspeicher (55296 bis 56295)

SCR # 29: nicht verwendet.

SCR # 30 bis 32: Painter-Package (tastaturgesteuertes Zeichenprogramm).

SCR # 33 bis 39: nicht verwendet.

SCR # 40 bis 42: Shape-Package (Teil 1).

SCR # 43 bis 44: nicht verwendet.

SCR # 45 bis 46: Shape-Package (Teil 2).

SCR # 47 bis 49: nicht verwendet.

SCR # 50: String-Package, grundsätzliche Definitionen.

SCR # 51 bis 54: nicht verwendet.

SCR # 55 bis 57: String-Package, zusätzliche Funktionen.

SCR # 58 bis 59: nicht verwendet.

SCR # 60 bis 61: Turtle-Package, stellt in nur zwei Screens alle für die Turtle-Grafik notwendigen Befehle zusammen.

SCR # 62 bis 69: nicht verwendet.

SCR # 70 bis 77: Nun lernen Sie die Wörter PLOT und LINK auch als Primitive (für C 64) kennen. Dazu benötigen Sie das Assembler-Vokabular Ihres FIG-Forth und ein ganzes Stück Arbeit, um die Screens einzugeben. Dafür haben Sie dann aber auch einen sehr schnellen LINK-Algorithmus.

XFORM und YFORM passen die Koordinaten an, die »außerhalb« liegen, das heißt, die Werte $0 \geq x \geq 319$ oder $0 \geq y \geq 199$, so daß Linien, die auf einer Seite herauslaufen, auf der anderen wieder auftauchen. (»Wrap-Around«.) Der fortgeschrittene Programmierer kann sich diese Wörter auch so abändern, daß die Linie am Rand einfach abbricht, also einen maximalen Wert nicht überschreitet (»Clipping«). Bemerkenswert ist vielleicht, daß UNLINK den Code von LINK abändert, indem es die Adresse von UNPLOTCODE in den Code-Body von PLOTIT schreibt und dann einfach LINK aufruft. Dies ist eine Programmiertechnik, die sehr mit Vorsicht zu genießen ist!

Eine Liste aller Befehle mit ihren Wirkungen ist in der Tabelle (rechts) enthalten.

Aufbauend auf dem Grundprogramm können Sie eine Vielzahl eigener Programme verwirklichen. Als Anregung einige Beispiele, wie das tastaturgesteuerte Zeichenprogramm (PAINTER), das Speichern und Kopieren von beliebigen Bildschirmausschnitten (SHAPES), der Sprite-Generator (SHAPER) oder die Logo entlehnte Schildkröte (TURTLE). Schließlich können Sie auch Texte (STRINGs) aus dem Character-ROM auslesen und in die HiRes einbringen.

(Andreas Carl/ev)

(1) HiRes

HIBASE	(- 8192)	Adresse des ersten Bytes
SCROFF	(-)	ausschalten des Video-Chip
SCRON	(-)	einschalten des Video-Chip
HION	(-)	umschalten auf Grafikbildschirm
HIOFF	(-)	umschalten auf Arbeitsbildschirm
HION?	(-f)	Grafikbildschirm eingeschaltet?
HFILL	(n-)	füllt die Grafikseite mit dem Byte n
HCLEAR	(-)	löscht den Grafikbildschirm
HCOL	(n-)	wählt Zeichenfarbe $0 \leq n \leq 15$
CFILL	(b c -)	füllt den Farbspeicher mit der Hintergrundfarbe b und der Zeichenfarbe c
ADR	(x y - a)	Wandelt die Koordinaten x, y in die Bitmap-Adresse a um
PLOT	(x y -)	setzt Zeichenpunkt
UNPLOT	(x y -)	löscht Zeichenpunkt
?PLOT	(x y - f)	Zeichenpunkt gesetzt?
XFORM	:;	Handler für Koordinaten, die »außerhalb« liegen;
LINK	(x_0y_0 x_1y_1 -)	verbindet zwei Punkte P_0 und P_1
UNLINK	(x_0y_0 x_1y_1 -)	löscht die Verbindungslinie zweier Punkte P_0 und P_1
BREAK	(n -)	definiert die Länge der Teile von gebrochenen Linien. Ist zu n = 30000 initialisiert
NGON	(x_0y_0..x_{n-1}..y_{n-1} n -)	verbindet n Punkte P_0 bis P_{n-1}
POLYGON	(x_0y_0..$x_{n-1}y_{n-1}$ n -)	verbindet n Punkte P_0 bis P_{n-1} zu einem geschlossenen Kurvenzug
TRIC	(x_0y_0 x_1y_1 x_2y_2 -)	zeichnet ein Dreieck $P_0P_1P_2$
REC	(x_0y_0 ab—)	zeichnet ein Rechteck mit der Länge und der Höhe b
QUADRAT	(x_0y_0 a -)	zeichnet ein Quadrat mit der Kantenlänge a
CIRCLE	(x_0y_0 r d α -)	zeichnet einen Kreis um den Mittelpunkt P_0 mit dem Radius r in Schritten von je d α Grad
ARC	(x_0y_0 r α_0 α_1 dα-)	zeichnet einen Teilkreis von α_0 Grad bis α_1 Grad
ANGL	x_0y_0 r α -)	zeichnet Radius eines Kreises zum Winkel α
EXCENTER	(n -)	bestimmt die Exzentrizität für CIRCLE, ARC und ANGL; n= 100 bedeutet $r_x : r_y = 1$
SEQ name	(-)	legt Filenamen für folgende Diskettenoperationen fest
.SEQ	(-)	zeigt Filenamen an
SWRITE	(-)	öffnet sequentielles Schreibfile
SREAD	(-)	öffnet sequentielles Lesefile
SCLOSE	(-)	schließt sequentielles File
TO	(-)	schreibt Byte b in offenes File
TOS	(a n -)	schreibt n Bytes ab der Adresse a in offenes File
FROM	(- b)	liest Byte b ein
FROMS	(a n -)	liest n Bytes ein und speichert sie ab der Adresse a
HIWRITE	(-)	schreibt 8 KByte HiRes plus 3 KByte Farbinformation in sequentielles File
HIREAD	(-)	liest HiResgrafik aus sequentiellem File ein
SHAPEWRITE	(-)	speichert Shapes oder Sprites in sequentielles File
SHAPEREAD	(-)	liest Shape oder Sprite

(2) Multi-Color-Grafik

alle Befehle wie (1) und zusätzlich:

MON	(-)	schaltet auf Multi-Color-Mode um
MOFF	(-)	schaltet auf Normal-Mode um
MFLAG	(- f)	Multi-Color-Mode eingeschaltet?
CFILL	(b $c_1c_2c_3$ -)	füllt die Farbspeicher
MCOL	($c_1c_2c_3$ -)	wählt die drei Zeichenfarben ($0 \leq c \leq 15$)
HCOL	(n -)	wählt aktuelle Farbe für PLOT etc. (n = 1,2,3)

Die Befehle des Grafik-Pakets

Listing »Turtle-Grafik« in Forth (Fortsetzung)

```
SCR # 18
 0         ( CONT. )
 1   ' ADR,H     CFA VARIABLE 'ADR
 2   ' ?PLOT,H  CFA VARIABLE '?PLOT
 3   ' PLOT,H   CFA VARIABLE 'PLOT
 4   ' UNPLOT,H CFA VARIABLE 'UNPLOT
 5   ' HCOL,H   CFA VARIABLE 'HCOL
 6   ' CFILL,H  CFA VARIABLE 'CFILL
 7
 8 : ADR 'ADR @ EXECUTE ;
 9 : ?PLOT '?PLOT @ EXECUTE ;
10 : PLOT 'PLOT @ EXECUTE ;
11 : UNPLOT 'UNPLOT @ EXECUTE ;
12 : -PLOT VPLOT @ VBREAK @ / 2 MOD
13         0= IF PLOT ELSE UNPLOT ENDIF VPLOT DUP @ 1+ SWAP ! ;
14 : HCOL 'HCOL @ EXECUTE ;
15 : CFILL 'CFILL @ EXECUTE ;                            -->

SCR # 19
 0         ( CONT. )
 1
 2
 3 : MCON 53270 C@ 16 OR 53270 C! ;
 4 : MCOFF 53270 C@ 239 AND 53270 C! ;
 5
 6 : MON MCON  ' PLOT,M CFA 'PLOT !  ' UNPLOT,M CFA 'UNPLOT !
 7              ' HCOL,M CFA 'HCOL !  ' CFILL,M  CFA 'CFILL  !
 8              ' ADR,M  CFA 'ADR  !  ' ?PLOT,M  CFA '?PLOT  !
 9              1 ' MFLAG ! ;
10 : MOFF MCOFF ' PLOT,H CFA 'PLOT !  ' UNPLOT,H CFA 'UNPLOT !
11               ' HCOL,H CFA 'HCOL !  ' CFILL,H  CFA 'CFILL  !
12               ' ADR,H  CFA 'ADR  !  ' ?PLOT,H  CFA '?PLOT  !
13              0 ' MFLAG ! ;
14
15

SCR # 20
 0         ( LINK )
 1
 2     0 VARIABLE DX    0 VARIABLE DY    0 VARIABLE NSTEP
 3
 4 : TO    DUP 0< IF 1 - -1 NSTEP ! ELSE 1 + 1 NSTEP ! ENDIF ;
 5
 6 : LINX DO DUP I DY @ DX @ */ +
 7         ROT DUP I + ROT -PLOT SWAP NSTEP @ +LOOP 2DROP ;
 8 : LINY DO DUP I DX @ DY @ */ +
 9         ROT DUP I + ROT SWAP -PLOT SWAP NSTEP @ +LOOP 2DROP ;
10
11 : LINK INIPLOT ROT >R I - DY ! SWAP >R I - DX !
12         R> R> DX @ ABS DY @ ABS > IF DX @ TO 0 LINX
13                          ELSE SWAP DY @ TO 0 LINY ENDIF ;
14 : UNLINK VBREAK @ >R 0 VBREAK ! LINK R> VBREAK ! ;
15

SCR # 21
 0         ( SINUS )
 1 0 VARIABLE SINA 175 , 349 , 523 , 698 , 872 , 1045 , 1219 , 1392
 2  , 1564 , 1763 , 1908 , 2079 , 2250 , 2419 , 2588 , 2756 , 2924
 3  , 3090 , 3256 , 3420 , 3584 , 3746 , 3907 , 4067 , 4226 , 4384
 4  , 4540 , 4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 , 5736
 5  , 5878 , 6018 , 6157 , 6293 , 6428 , 6561 , 6691 , 6820 , 6947
 6  , 7071 , 7193 , 7314 , 7431 , 7547 , 7660 , 7771 , 7880 , 7986
 7  , 8090 , 8192 , 8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829
 8  , 8910 , 8988 , 9063 , 9135 , 9205 , 9272 , 9336 , 9397 , 9455
 9  , 9511 , 9563 , 9613 , 9659 , 9703 , 9744 , 9781 , 9816 , 9848
10  , 9877 , 9903 , 9925 , 9945 , 9962 , 9976 , 9986 , 9994 , 9998
11  , 10000 ,
12
13 : SIN@ 2 * SINA + @ 10000 SWAP ;
14 : COS@ 90 SWAP - SIN@ ;
15

SCR # 22
 0         ( CIRCLE )
 1
 2     0 VARIABLE DA    0 VARIABLE RAD    100 VARIABLE EXC
 3
 4 : CNORM BEGIN DUP   0 < WHILE 360 + REPEAT
 5         BEGIN DUP 360 > WHILE 360 - REPEAT ;
 6
 7 : SIN CNORM DUP DUP ABS / SWAP ABS DUP 90 < IF SIN@ ELSE
 8       DUP 180 < IF 180 SWAP - SIN@ ELSE
 9       DUP 270 < IF 180 - SIN@ -1 * ELSE
10       360 SWAP - SIN@ -1 * ENDIF ENDIF ENDIF ROT * ;
11 : COS 90 SWAP - SIN ;
12
13 : EPS EXC @ 100 */ ;
14 : EXCENTER EXC ! ;                                    -->
15

SCR # 23
 0         ( CONT. )
 1
 2   0 VARIABLE CX     0 VARIABLE CY
 3
 4 : CIRC RAD @ OVER SIN SWAP */ RAD @ ROT COS SWAP */ ;
 5
 6 : ARC DA ! 2DUP > IF DA DUP @ -1 * SWAP ! ENDIF
 7       >R >R RAD ! CY ! CX ! R> R> SWAP
 8       DO I CIRC CX @ + SWAP EPS CY @ + R> DA @ + >R
 9          I CIRC CX @ + SWAP EPS CY @ + R> DA @ - >R
10          LINK DA @ +LOOP ;
11 : CIRCLE 0 360 ROT ARC ;
12 : ANGL    >R RAD ! 2DUP R> CIRC >R + SWAP R> + SWAP LINK ;
13
14
15

SCR # 24
 0         ( POLYGON )
 1
 2   0 VARIABLE B     0 VARIABLE L
 3
 4 : GON      1 DO >R >R 2DUP R> R> 2SWAP LINK LOOP ;
 5 : NGON     GON 2DROP ;
 6 : POLYGON ROT ROT >R DUP I SWAP >R ROT GON R> R> LINK ;
 7 : TRIC 3 POLYGON ;
 8 : REC  1 - B ! 1 - L !
 9         2DUP L @ ROT + SWAP
10         2DUP B @ +
11         2DUP L @ ROT SWAP - SWAP
12         2DUP B @ -
13         5 NGON ;
14 : QUADRAT DUP REC ;
15

SCR # 25
 0         ( SEQUENTIELLE FILES )
 1
 2 0 VARIABLE SF 20 ALLOT
 3
 4 :  SEQ 34 WORD HERE SF 20 CMOVE ;
 5 : .SEQ SF COUNT TYPE ;
 6
 7 : NAM,S,X SF COUNT + DUP ASCII , SWAP C!
 8                  1+ DUP ASCII S SWAP C!
 9                  1+ DUP ASCII , SWAP C!
10                  1+                  C!
11              SF DUP C@ 4 + SWAP C! ;
12 : NAM,S,W ASCII W NAM,S,X ;
13 : NAM,S,R ASCII R NAM,S,X ;
14                                                       -->
15

SCR # 26
 0         ( CONT. )
 1
 2   HEX   0 VARIABLE SPSAVE
 3
 4 : OFFSPRITE D015 C@ SPSAVE C! 0 D015 C! ;
 5 : OLDSPRITE SPSAVE C@ D015 C! ;
 6
 7 : SWRITE OFFSPRITE CR ." WRITING " .SEQ
 8         SYSTEM NAM,S,W SF COUNT SWAP 100 /MOD FFBD SYS
 9         3DROP SF DUP C@ 4 - SWAP C!
10         2 8 2 FFBA SYS FFC0 SYS 3DROP
11         0 2 0 FFC9 SYS 3DROP ;
12
13 : TO SYSTEM 0 0 FFD2 SYS 3DROP ;
14 : TOS 0 DO DUP I + C@ TO LOOP DROP ;                  -->
15

SCR # 27
 0         ( CONT. )
 1
 2
 3 : SREAD OFFSPRITE CR ." READING " .SEQ
 4         SYSTEM NAM,S,R SF COUNT SWAP 100 /MOD FFBD SYS
 5         3DROP SF DUP C@ 4 - SWAP C!
 6         2 8 2 FFBA SYS FFC0 SYS 3DROP
 7         0 2 0 FFC6 SYS 3DROP ;
 8
 9 : SCLOSE OLDSPRITE SYSTEM 2 0 0 FFC3 SYS FFCC SYS 3DROP ;
10
11 : FROM SYSTEM 0 0 0 FFCF SYS 2DROP ;
12 : FROMS 0 DO FROM OVER I + C! LOOP DROP ;
13
14                                  DECIMAL
15

SCR # 28
 0         ( WRITE/READ )
 1
 2
 3 : HIWRITE SWRITE 8192 DUP TOS 53281 C@ TO
 4                  HION? IF 1024 ELSE 4096 ENDIF 1000 TOS
 5                  55296 1000 TOS SCLOSE ;
 6 : HIREAD SREAD 8192 DUP FROMS FROM 53281 C!
 7                  HION? IF 1024 ELSE 4096 ENDIF 1000 FROMS
 8                  55296 1000 FROMS SCLOSE ;
 9
10 : SHAPEWRITE SWRITE + TOS   SCLOSE ;
11 : SHAPEREAD SREAD  + FROMS SCLOSE ;
12
13
14
15
```

```
SCR # 30
 0          ( PAINTER )
 1   0 VARIABLE OLD
 2 : REPON 128 650 C! ;
 3 : REPOFF 0 650 C! ;
 4 : KILL 2DUP UNPLOT ;
 5 : PLOTOVER 2DUP ADR OLD @ SWAP C! ;
 6 : DRAW ;
 7    ' DRAW     CFA VARIABLE LASTVECTOR
 8    ' PLOTOVER CFA VARIABLE PENMODE
 9 : UP/DOWN PENMODE @ LASTVECTOR @ PENMODE ! LASTVECTOR ! ;
10 : ?DOWN ' PLOTOVER CFA PENMODE @ = ;
11 : SETMODUS ?DOWN IF LASTVECTOR ! ELSE PENMODE ! ENDIF ;
12 : DODRAW  ' DRAW  CFA SETMODUS ;
13 : DOKILL ' KILL CFA SETMODUS ;
14 : LASTPLOT >R LASTVECTOR @ EXECUTE R> ;
15 : NEXTPLOT 2DUP ADR C@ OLD ! 2DUP PLOT ;                  -->

SCR # 31
 0          ( CONT. )
 1 : SLOPE BEGIN KEY DUP 13 = 0= WHILE LASTPLOT
 2          DUP 72 = IF >R SWAP 1+ SWAP R> ENDIF
 3          DUP 78 = IF >R 1+ SWAP 1+ SWAP R> ENDIF
 4          DUP 66 = IF >R 1+ R> ENDIF
 5          DUP 86 = IF >R 1+ SWAP 1 - SWAP R> ENDIF
 6          DUP 70 = IF >R SWAP 1 - SWAP R> ENDIF
 7          DUP 82 = IF >R 1 - SWAP 1 - SWAP R> ENDIF
 8          DUP 84 = IF >R 1 - R> ENDIF
 9          DUP 89 = IF >R 1 - SWAP 1+ SWAP R> ENDIF
10         DUP  71 = IF UP/DOWN ENDIF
11         DUP 133 = IF 3 HCOL DODRAW ENDIF
12         DUP 134 = IF 2 HCOL DODRAW ENDIF
13         DUP 135 = IF 0 HCOL DODRAW ENDIF
14             136 = IF DOKILL ENDIF
15         NEXTPLOT REPEAT DROP ;                    -->

SCR # 32
 0          ( CONT. )
 1
 2 : PAINT 2DUP PLOT
 3          ' DRAW     CFA LASTVECTOR !
 4          ' PLOTOVER CFA PENMODE !
 5          REPON SLOPE REPOFF ;
 6
 7
 8   (       R T Y                 F1
 9                                 F3
10          F G H                  F5
11                                 F7
12            V B N    RETURN             )
13
14
15

SCR # 40
 0          ( SHAPES )
 1
 2   0 VARIABLE PARITY
 3
 4 : SHAPE <BUILDS OVER , DUP , * ALLOT
 5          DOES> >R R 4 + R @ R> 2+ @ ;
 6 : CLEAR * 0 DO 0 OVER I + C! LOOP DROP ;
 7 : ?S OVER HERE ! * 0 DO I HERE @ MOD 0=
 8       IF CR ENDIF DUP I + C@ 4 .R LOOP DROP ;
 9 : DIM ROT DROP ;
10 : ZAHL? 0 IN ! TIB @ 10 EXPECT
11          32 WORD HERE NUMBER DROP ;
12 : INPUT OVER PARITY ! * 0 DO I PARITY @ MOD 0= IF CR ENDIF
13                            ZAHL? OVER I + C! LOOP DROP ;
14
15                                                          -->

SCR # 41
 0          ( CONT. )
 1     0 VARIABLE SX    0 VARIABLE SY
 2
 3 : >BIT 8 0 DO 2 * >R I 256 / R> 255 AND LOOP DROP ;
 4 : 8SWAP 8 0 DO HERE I + C! LOOP
 5          8 0 DO HERE I + C@ LOOP ;
 6 : PUT SY ! SX ! DY ! DX !  0 SWAP
 7       DX @ DY @ * 0 DO DUP I + C@ >BIT 8SWAP
 8                          8 0 DO SX @ SY @ ROT
 9                               IF PLOT ELSE UNPLOT ENDIF
10                               1 SX +! LOOP
11                        >R 1+ DUP DX @ MOD 0= IF 1 SY +!
12                                            -8 DX @ * SX +!
13                                        ENDIF R> LOOP 2DROP ;
14
15                                              45 ?LOAD

SCR # 42
 0          ( SHAPER )
 1   16 VARIABLE PX   16 VARIABLE PY
 2
 3 : 3DUP >R OVER OVER R ROT ROT R> ;
 4 : FRAME ROT DROP 2+ SWAP 8 * 2+ SWAP 15 15 2SWAP REC ;
 5 : DEFSHAPE OVER HERE ! * 16 PX ! 15 PY !
 6            0 DO I HERE @ MOD 0= IF 1 PY +! 16 PX !
 7                                     ELSE 8 PX +! ENDIF
 8                 PX @ PY @ ADR C@ OVER C! 1+ LOOP DROP ;
 9 : SHAPER HCLEAR 3DUP FRAME 17 17 PAINT
10          2DROP DEFSHAPE ;
11
12 : SPRITE 2040 + C@ 64 * 3 21 ;
13 : !POINTER 8 0 DO 48 I + 2040 I + C! LOOP ;
14
15                                                 46 ?LOAD

SCR # 45
 0          ( MPUT )
 1
 2 : MPUT SY ! SX ! DY ! DX !  0 SWAP
 3       DX @ DY @ * 0 DO DUP I + C@ >BIT 8SWAP
 4                        4 0 DO 2 * * DUP
 5                               IF HCOL SX @ SY @ PLOT
 6                               ELSE DROP SX @ SY @ UNPLOT ENDIF
 7                               1 SX +! LOOP
 8                        >R 1+ DUP DX @ MOD 0= IF 1 SY +!
 9                                            -4 DX @ * SX +!
10                                        ENDIF R> LOOP 2DROP ;
11
12
13
14
15

SCR # 46
 0          ( MSHAPER )
 1
 2
 3 : MFRAME ROT DROP 2+ SWAP 4 * 2+ SWAP 7 15 2SWAP REC ;
 4 : DEFMSHAPE OVER HERE ! * 16 PX ! 15 PY !
 5            0 DO I HERE @ MOD 0= IF 1 PY +! 16 PX !
 6                                     ELSE 8 PX +! ENDIF
 7                 PX @ PY @ ADR,H C@ OVER C! 1+ LOOP DROP ;
 8
 9 : MSHAPER HCLEAR 3DUP MFRAME 9 17 PAINT
10          2DROP DEFMSHAPE ;
11
12
13
14
15

SCR # 50
 0          ( STRING-PACKAGE )
 1
 2 :       $ <BUILDS 78 ALLOT DOES> ;
 3
 4 : INPUT$ 0 IN ! TIB @ 80 EXPECT 34 WORD
 5          HERE SWAP 80 CMOVE ;
 6 :     =$ 34 WORD HERE SWAP 80 CMOVE ; IMMEDIATE
 7
 8 :      .$ COUNT TYPE ;
 9 :    LEN COUNT SWAP DROP ;
10
11
12
13
14
15

SCR # 55
 0          ( @CHAR )
 1
 2 CODE @CHAR 254 # LDA, 56334 AND, 56334 STA,
 3            251 # LDA, 1 AND, 1 STA,
 4
 5            BOT LDA, N 1 - STA,
 6            BOT 1+ LDA, N STA,
 7            0 # LDA, BOT 1+ STA, TAY,
 8            N 1 - )Y LDA, BOT STA,
 9
10            4 # LDA, 1 ORA, 1 STA,
11            1 # LDA, 56334 ORA, 56334 STA,
12
13            NEXT JMP, END-CODE
14
15

SCR # 56
 0          ( ASCII-WANDLER )
 1
 2
 3 : A>S DUP 128 AND IF 127 AND 64 OR ELSE
 4       DUP  64 AND 0= IF ELSE
 5       DUP  32 AND IF 95 AND ELSE
 6            63 AND ENDIF ENDIF ENDIF ;
 7
 8
 9
10
11
12
13
14
15
```



```
SCR # 57
 0        ( PUT@ )
 1
 2     53248 CONSTANT CHARBASE
 3         0 VARIABLE CHARACTER 6 ALLOT
 4
 5 : CHR 8 * CHARBASE +
 6       8 0 DO DUP I + @CHAR CHARACTER I + C!
 7           LOOP DROP CHARACTER 1 8 ;
 8
 9         0 VARIABLE CHX   0 VARIABLE CHY
10
11 : PUT@ CHY ! CHX ! COUNT 0 DO DUP I + C@ A>S CHR
12                                CHX @ I 8 * + CHY @ PUT
13                             LOOP DROP ;
14
15

SCR # 60
 0        ( TURTLE )
 1
 2     0 VARIABLE HEADING    1 VARIABLE PENSTATE
 3
 4
 5 : RIGHT HEADING +! ;
 6 : LEFT  -1 * RIGHT ;
 7 : MOVE 2DUP >R >R 2SWAP R> R>
 8        PENSTATE @ IF LINK ELSE 2DROP 2DROP ENDIF ;
 9 : FORWARD RAD ! 2DUP HEADING @ 90 - CIRC >R + SWAP R> +
10           SWAP MOVE ;
11 : BACK 180 RIGHT FORWARD 180 LEFT ;
12
13
14                                                     -->
15

SCR # 61
 0        ( CONT. )
 1
 2 : PENDOWN 1 PENSTATE ! ;
 3 : PENUP 0 PENSTATE ! ;
 4 : SETHEADING HEADING ! ;
 5 : SETX OVER MOVE ;
 6 : SETY >R OVER R> MOVE ;
 7 : SETXY MOVE ;
 8 : HOME 0 SETHEADING 160 100 MOVE ,
 9 : TURTLE HCLEAR HION 0 SETHEADING MFLAG IF 11 1 0 6 CFILL
10          80 100 ELSE 11 1 CFILL 160 100 ENDIF ;
11
12   : FD FORWARD ;         : BK BACK ;
13   : RT RIGHT ;           : LT LEFT ;
14   : PD PENDOWN ;         : PU PENUP ;
15   : SETH SETHEADING ;

SCR # 70
 0        ( FORM )
 1
 2 CODE XFORM BEGIN, SEC, BOT LDA, 64 # SBC, BOT STA,
 3                        BOT 1+ LDA, 1 # SBC, BOT 1+ STA,
 4                 0< UNTIL,
 5             BEGIN, CLC, BOT LDA, 64 # ADC, BOT STA,
 6                        BOT 1+ LDA, 1 # ADC, BOT 1+ STA,
 7                 0< NOT UNTIL, RTS, END-CODE
 8
 9 CODE YFORM BEGIN, SEC, BOT LDA, 200 # SBC, BOT STA,
10                        BOT 1+ LDA, 0 # SBC, BOT 1+ STA,
11                 0< UNTIL,
12             BEGIN, CLC, BOT LDA, 200 # ADC, BOT STA,
13                        BOT 1+ LDA, 0 # ADC, BOT 1+ STA,
14                 0< NOT UNTIL, RTS, END-CODE
15                                                     -->

SCR # 71
 0        ( QUICK-PLOT )
 1   ASSEMBLER HEX
 2   0     VARIABLE X        0 VARIABLE Y          N CONSTANT XL
 3   N 1 + CONSTANT XH    N 2 + CONSTANT SUML    N 3 + CONSTANT SUMH
 4   N 1 - CONSTANT FE
 5 CODE PLOTBODY  ' YFORM JSR, INX, INX, ' XFORM JSR, DEX, DEX,
 6                XSAVE STX, BOT LDY,
 7                BOT 2+ LDA, PHA, BOT 3 + LDA, TAX, PLA,
 8            XL STA, XH STX, TYA, F8 # AND, FE STA, SUML STA,
 9            0 # LDA, SUMH STA, SUML ASL, SUMH ROL, SUML ASL,
10            SUMH ROL, CLC, SUML LDA, FE ADC, SUML STA, SUMH LDA,
11            0 # ADC, SUMH STA, SUML ASL, SUMH ROL, SUML ASL,
12            SUMH ROL, SUML ASL, SUMH ROL, TYA, 7 # AND, CLC,
13            SUML ADC, SUML STA, SUMH LDA, 0 # ADC, SUMH STA,
14            CLC, XL LDA, F8 # AND, SUML ADC, SUML STA, XH LDA,
15            SUMH ADC, SUMH STA, CLC, 0 # LDA, SUML ADC, -->

SCR # 72
 0        ( CONT. )
 1      SUML STA, 20 # LDA, SUMH ADC, SUMH STA, XL LDA, 7 # AND,
 2      7 # EOR, TAX, 1 # LDA,
 3      BEGIN, .A ASL, DEX, 0< UNTIL, .A ROR,
 4      0 # LDY, RTS, END-CODE
 5
 6 CODE PLOTCODE   ' PLOTBODY JSR, SUML )Y ORA,
 7   SUML )Y STA, XSAVE LDX,
 8   INX, INX, INX, INX, RTS, END-CODE
 9 CODE UNPLOTCODE ' PLOTBODY JSR, FF # EOR, SUML )Y AND,
10   SUML )Y STA, XSAVE LDX, INX, INX, INX, INX, RTS, END-CODE
11 CODE ?PLOT  ' PLOTBODY JSR, SUML )Y AND, XSAVE LDX,
12   INX, INX, BOT STA, 0 # LDA, BOT 1+ STA, NEXT JMP, END-CODE
13 CODE ADR ' PLOTBODY JSR, XSAVE LDX, INX, INX,
14          SUML LDA, BOT STA, SUMH LDA, BOT 1+ STA, NEXT JMP,
15          END-CODE    -->

SCR # 73
 0        ( CONT. )
 1
 2 CODE (PLOT) ' PLOTCODE JSR, SEC, SUMH LDA, 20 # SBC, SUMH STA,
 3          SUML LDA, F8 # AND, SUML STA, CLC,
 4 SUMH ROR, SUML ROR, SUMH ROR, SUML ROR, SUMH ROR,
 5 SUML ROR, 4 # LDA, SUMH ADC, SUMH STA, SUML )Y LDA, OF # AND,
 6 SUML )Y STA, COL LDA, .A ASL, .A ASL, .A ASL, .A ASL, SUML )Y
 7 ORA, SUML )Y STA, RTS, END-CODE
 8
 9 CODE  QPLOT  ' PLOTCODE   JSR, NEXT JMP, END-CODE
10 CODE UNPLOT ' UNPLOTCODE JSR, NEXT JMP, END-CODE
11 CODE   PLOT ' (PLOT) JSR, NEXT JMP, END-CODE
12
13 : INIPLOT ;   : -PLOT PLOT ;    : VBREAK ;    : BREAK DROP ;
14
15   DECIMAL

SCR # 74
 0        ( QUICK-LINK )               HEX
 1
 2   0 VARIABLE XO    0 VARIABLE YO    0 VARIABLE X1    0 VARIABLE Y1
 3   0 VARIABLE OF    0 VARIABLE CT    0 VARIABLE DX    0 VARIABLE DY
 4   0 VARIABLE IX    0 VARIABLE IY    0 VARIABLE AX    0 VARIABLE AY
 5
 6
 7 CODE PLOTIT XSAVE LDX,
 8             YO LDA, BOT    STA, YO 1+ LDA, BOT 1+  STA,
 9             XO LDA, BOT 2+ STA, XO 1+ LDA, BOT 3 + STA,
10             ' (PLOT) JSR, RTS, END-CODE
11
12
13
14                                                     -->
15

SCR # 75
 0        ( CONT. )
 1 CODE +STEP SEC, OF    LDA, DX    SBC, OF    STA,
 2                 OF 1+ LDA, DX 1+ SBC, OF 1+ STA,
 3      AY LDA, 0< IF, SEC, XO    LDA, 1 # SBC, XO    STA,
 4                          XO 1+ LDA, 0 # SBC, XO 1+ STA, ELSE,
 5                     CLC, XO    LDA, AY  ADC, XO    STA,
 6                          XO 1+ LDA, 0 # ADC, XO 1+ STA, ENDIF,
 7      IY LDA, 0< IF, SEC, YO    LDA, 1 # SBC, YO    STA,
 8                          YO 1+ LDA, 0 # SBC, YO 1+ STA, ELSE,
 9                     CLC, YO    LDA, IY  ADC, YO    STA,
10                          YO 1+ LDA, 0 # ADC, YO 1+ STA, ENDIF,
11      RTS, END-CODE
12
13 : X<->Y DX @ DY @ DX ! DY !
14        IX @ AY ! IY @ AX !
15        0 DUP IX ! IY ! ;                            -->

SCR # 76
 0        ( CONT. )
 1 CODE LOP
 2 IX LDA, 0< IF, SEC, XO LDA, 1 # SBC, XO STA, XO 1+ LDA,
 3 0 # SBC, XO 1+ STA, ELSE, CLC, XO ADC, XO STA, XO 1+ LDA,
 4 0 # ADC, XO 1+ STA, ENDIF,
 5 AX LDA, 0< IF, SEC, YO LDA, 1 # SBC, YO STA, YO 1+ LDA,
 6 0 # SBC, YO 1+ STA, ELSE, CLC, YO ADC, YO STA, YO 1+ LDA,
 7 0 # ADC, YO 1+ STA, ENDIF,
 8 CLC, OF LDA, DY ADC, OF STA, OF 1+ LDA, DY 1+ ADC, OF 1+ STA,
 9 CT INC, 0= IF, CT 1+ INC, ENDIF,
10
11 OF 1+ LDA, DX 1+ CMP, CS IF, 0= NOT IF, ' +STEP JSR, ELSE,
12 DX LDA, OF CMP, CS NOT IF, ' +STEP JSR, ENDIF, ENDIF, ENDIF,
13 ' PLOTIT JSR, RTS, END-CODE
14                                                     -->
15

SCR # 77
 0        ( CONT. )
 1 CODE LINKCODE DEX, DEX, DEX, DEX, XSAVE STX, ' PLOTIT JSR,
 2               BEGIN, ' LOP JSR,
 3               0 # LDX, CT 1+ LDA, DX 1+ CMP, CS NOT IF, INX,
 4               ELSE, DX LDA, CT CMP, CS IF, INX, ENDIF, ENDIF,
 5               TXA, 0= UNTIL, XSAVE LDX, INX, INX, INX, INX,
 6               NEXT JMP, END-CODE
 7
 8 : LINK Y1 ! X1 ! YO ! XO ! 0 DUP AY ! AX !
 9       X1 @ XO @ 2DUP > IF 1 IX ! ELSE -1 IX ! SWAP ENDIF - DX !
10       Y1 @ YO @ 2DUP > IF 1 IY ! ELSE -1 IY ! SWAP ENDIF - DY !
11       DX @ DY @ > 0= IF X<->Y ENDIF DX @ 2 / OF ! 1 CT !
12       LINKCODE ;
13 : UNLINK ' UNPLOTCODE ' PLOTIT 17 + ! LINK
14          ' (PLOT)     ' PLOTIT 17 + ! ;          DECIMAL
15
```

Listing »Turtle-Grafik« in Forth (Schluß)

x-pert, ein Mini-Experten-System in Forth

x-pert ist ein Mini-Experten-System, das bis zu 2512 Menü- oder Antwort-Blöcke mit 31 Byte Text und bis zu je zehn Menü-Verzweigungen zuläßt. Es bestehen Möglichkeiten zur Erweiterung, Änderung, zum Löschen und zur Mehrfachnutzung von Menüs und Antworten. Der Editor ist auch direkt einsetzbar.

Der Versuch, ein Experten-System auf einem Heimcomputer zu realisieren, scheitert meist an der geringen Speicherkapazität oder an den extrem langen Reaktionszeiten des Diskettenlaufwerks beim Suchen von Datensätzen. Einen vertretbaren Kompromiß gestattet die Programmiersprache Forth, die durch ihr virtuelles Speicherkonzept fast die gesamte Diskette als RAM nutzt. Ein reines Assembler-Programm erlaubt zwar kürzere Verarbeitungszeiten, hält das System aber nicht für Modifikationen und Erweiterungen offen.

In der hier vorgeschlagenen Lösung für den Commodore 64, enthält jeder Datenblock 31 Zeichen Text und bis zu zehn Verzweigungen. Eine Modifikation des Forth-Wortes »r/w« (scr # 8,line # 1) erlaubt es, maximal 162 screens auf eine Diskette zu schreiben. So stehen insgesamt 2512 Datenblöcke zur Verfügung. Um einen direkten Einblick und somit auch direkte Änderungsmöglichkeiten zu schaffen, werden alle Daten im ASCII-Code gespeichert, wobei die Nummern der Folgeblöcke als hexadezimale Zahlen erscheinen. So müssen auch alle Eingaben im hexadezimalen Modus erfolgen.

Neben den üblichen Variationen einen Entscheidungsbaum aufzubauen und zu verändern, erlaubt das vorgestellte System bestehende Menüs oder Antworten in neue Menüs einzufügen. Das führt zwar zu einer Platzersparnis, doch wächst damit die Gefahr der Verfilzung der Äste oder es führt zu Endlosschleifen bei der Darstellung der Baumstruktur.

Im folgenden sollen die neuen, von unserem Mini-System angebotenen Wörter erklärt werden (vergleiche Listing 1).

System-Start

System mit »7 load« von der Programm-Diskette laden. Danach wird eine Daten-Diskette angefordert.

Oder »x-pert« (= = >) eingeben (nach erstmaliger Compilation). Das System wird neu gestartet, ohne eventuell neue oder veränderte Daten aus dem Disk-Buffer zu übernehmen.

Beenden der Arbeit

Nach jeder Sicherung der Daten mit »save« oder »start« kann die Bearbeitung einer Diskette beziehungsweise einer Datei abgeschlossen werden. Das Leuchten der LED am Laufwerk ist ohne Bedeutung. Ein Abschluß mit »done« ist ebenfalls möglich.

Zwischensicherung ohne Protokollveränderung

`save ( ==>)`

Die Datenblöcke im Disketten-Puffer werden auf die Diskette geschrieben und die Bearbeitung vor dem aktuellen Menü fortgesetzt.

`done ( ==>)`

Hat die gleiche Wirkung wie »save«, ruft jedoch kein neues Menü auf.

Zwischensicherung mit Protokollneustart

`start ( ==>)`

Die Datenblöcke werden im Disketten-Puffer auf die Diskette geschrieben und die Bearbeitung mit neuer Protokollierung startet beim aktuellen Menü.

Neue Datendiskette einrichten

`new-disk ( ==>)`

Dieser Vorgang dauert etwa 25 Minuten. Beim Aufruf muß eine eingerichtete Datendiskette oder die Programmdiskette im Laufwerk sein. Bei Aufforderung legt man die neue Diskette ein. Es wird eine Titelzeile angefordert und dann mit dieser Datei gestartet.

Diskette aufräumen

`verify ( ==>)`

Ungenutzte Blöcke werden freigegeben. Dieser Vorgang wird mit zunehmender Anzahl von Daten immer zeitraubender. Man kann ihn auf dem Bildschirm beobachten. Ein fehlerhaftes Ändern mit dem Editor kann »verify« nicht immer korrigieren.

Menü anwählen

`↑(Blocknummer ==>)`

Das Menü beziehungsweise die Antwort mit der eingegebenen Blocknummer erscheint. Die Eingabe kann durch die Positionierung des Cursors unter die entsprechende Menüzeile erfolgen.

Rückwärtsschritt

`←( ==>)`

Es erfolgt die Rückkehr zum vorher aufgerufenen Menü. Die Ebene wird vermindert, im Protokoll sind jedoch alle Schritte festgehalten. »←« dient unter anderem dazu, zum Editor zurückzukehren.

Block zum Menü hinzufügen

Neueintrag

`x+   ( ==>)`

Sofern im Menü und auf der Diskette Platz ist, wird der durch Unterstrich angeforderte Text als neue Menüzeile an der ersten freien Stelle ins Menü eingefügt.

Bestehenden Block einfügen

`x+↑   (Blocknummer ==>)`

Der mit Blocknummer angesprochene Block wird in das aktuelle Menü eingefügt, sofern noch Platz vorhanden ist.

Menü zwischenschieben

`x←   ( Blocknummer ==>)`

Zwischen das aktuelle Menü und die angewählte Menüzeile fügt sich ein Menü ein. Der Text wird durch Unterstreichen angefordert.

Antwort löschen

`x-   ( Blocknummer ==>)`

Der Block wird freigegeben, sofern es sich um einen Antwortblock handelt.

Menü löschen

Ein Menü muß, bevor es gelöscht werden kann, mit

`x=   ( Blocknummer ===>)`

zum Antwortblock umgewandelt werden. Wenn keine Folgeblöcke vorliegen, wird das Menü laut Blocknummer zur Antwort.

Menü in Antwort wandeln

`x=    ( Blocknummer ==>)`

Wenn keine Folgeblöcke vorliegen, wandelt sich das Menü zur Antwort.

Text ändern

`corr   ( Blocknummer ==>)`

Der im angewählten Block gespeicherte Text kann erneut eingegeben werden.

Direktes Ändern mit dem Editor

Die reine ASCII-Darstellung ermöglicht alle Blöcke direkt zu ändern. Natürlich ist hierbei besondere Vorsicht geboten, da man leicht Schleifen oder falsche Verknüpfungen schaffen kann, die später kaum zu entwirren sind.

Hilfe zum Finden eines Blocks

`scr?    ( Blocknummer ==> )`

Es werden die Screen- und Zeilennummer zur hexadezimal eingegebenen Blocknummer dezimal angezeigt und in den Dezimal-Modus umgeschaltet.

Bildschirmprotokoll

`prot    ( ==> )`

Das Protokoll der bisherigen Abfragen wird nach Ebenen gestaffelt auf dem Bildschirm ausgegeben.

Ausdruck des Protokolls

`lprot    ( ==> )`

Geschachtelter Ausdruck der bisher aufgerufenen Menüs und Antworten.

Ausdruck der Gesamtdatei

`print    ( ==> )`

Alle belegten Datenblöcke werden in numerischer Reihenfolge mit Blocknummer, Text, Blockkennung und den Nummern der Folgeblöcke ausgedruckt.

Ausdruck der Baumstruktur

`baum    ( ==> )`

Beginnend ab dem aktuellen Menü als Stamm, wird die Baumstruktur in Form einer Gliederung ausgedruckt. Durch manuelle Menü-Verknüpfungen entstandene Schleifen führen zum Stack-Überlauf und Systemabsturz.

Beschreibung der einzelnen Datenblöcke

Bytes	Inhalt
00-1e	31 Byte Text
1f	Blocktyp-Kennung -: freier Block =: Antwort-Block >: Menü-Block v: Verwaltungs-Block
20-3e	zehn Folgeblock-Nummern jeweils drei ASCII-Zeichen als Hexadezimal-Zahl. Dieses Verfahren ist platzsparend und gestattet eine direkte Editierung.
3f	unbenutzt: blank
20	unbenutzt: Punkt

```
scr # 3
  0 *  MINI-EXPERTEN-SYSTEM: x-pert  *
  1 Verwaltungsblock           >.32062f407
  2 Version: 2.06
  3 (c) Peter Klinghardt Februar 1986
  4 ............................... -12
  5 ............................... -11
  6 ............................... -10
  7 Daten-Diskette einlegen !
  8 Leer-Diskette einlegen !
  9 Beliebige Taste druecken !
 10 noch Folge-Bloecke vorhanden
 11 ist kein Menue-Block
 12 ist kein Antwort-Block
 13 Diskette voll
 14 alle Menue-Zeilen belegt
 15 ............................... - 1
scr # 7
  0 (      Variable und Konstante       )
  1 forth definitions hex
  2 93 emit ." loading: x-pert" cr cr
  3    0 variable men     ( Nummer des aktuellen Menue-Blocks )
  4    0 variable vor
  5    0 variable nach
  6    0 variable addr    ( Zwischenspeicher )
  7    0 variable txt 20 allot ( Text-Bereich )
  8   60 constant anfang  ( erster Daten-Block )
  9  a2f constant ende    ( letzter Daten-Block )
 10   60 constant bis     ( letzter benutzter Daten-Block )
 11    0 variable en      ( Menue-Ebene )
 12 a000 constant ev      ( Menue-Eintraege Graphik-Bereich )
 13    0 variable vn      ( Protokoll-Ebene )
 14 b000 constant vv      ( Block-Nummern-Folge Graphik-Bereich)
 15 : m@ men @ ; : m! men ! ; -->
scr # 8
  0 (        Voreinstellungen           )
  1 0a2f 1f30 ! ( 162 Screens in r/w )
  2 ." ■" ( schwarze Schrift )
  3 : 1f+ 1f + ; : 21+ 21 + ;
  4 code prtof  ( ==>                   )
  5 ( Ausgabe auf Bildschirm            )
  6                  xsave stx, ffcc jsr,
  7                  xsave ldx, next jmp,
  8 end-code
  9 : prtoff cr prtof
 10 ; graphic &clear
 11 : x0  ( n ==>                       )
 12 ( Block n loeschen                  )
 13 1 ?enough        block dup 40 20 fill
 14               1f+ 2e2d swap ! update
 15 ; -->
scr # 9
  0 -10 message cr
  1 code prton  ( ==>                   )
  2 ( Ausgabe auf Drucker               )
  3 xsave stx,           4 # lda, 4 # ldx,
  4       7 # ldy, ffba jsr, ( SETLFS )
  5       0 # lda, ffbd jsr, ( SETNAM )
  6                ffc0 jsr, ( OPEN )
  7       4 # ldx, ffc9 jsr, ( CHKOUT )
  8     xsave ldx, next jmp,
  9 end-code
 10 : cn  ( c ==> b                     )
 11 ( ASCII-HEX in hex-Wert wandeln     )
 12 1 ?enough  dup 20 = if drop 0 else
 13     30 - dup a > if 7 - endif endif
 14 ;
 15 : 0> 0 > ; -->
```

Listing 1. »x-pert«, ein Mini-Expertensystem

Verwaltungsblock-Beschreibung
Blocknummer 31 <hex>

Bytes	Inhalt
00-1e	'Verwaltungsblock'
1f	'v'
20	'.'
21-23	erster Datenblock
24-26	letzter Datenblock
27-29	letzter belegter Block
2a...	Rest ungenutzt

x-pert ist in Forth 64 für den Commodore 64 geschrieben, sollte aber mit nur geringfügigen Modifikationen auf jedem anderen Computer unter FIG-Forth laufen.

Speziell für die Besitzer von Forth 64 sind in Listing 2 die Screens 40 bis 43 gedacht. Sie enthalten eine verbesserte Version der in Forth 64 fehlerhaften Wörter »triad« und eine verbesserte Version des Worts »dump«.

Diese Abänderung belegt keinen weiteren Speicherplatz, da sie in das bestehende System hineingeschrieben wird.

Allerdings ist sie daher auch wirklich nur mit Forth 64 zu verwenden, was jedoch nicht weiter schlimm ist, da diese Fehler ja ebenfalls spezifisch sind.

(Peter Klinghardt/ev)



```
scr # 10
  0 : nc   ( n ==> c                        )
  1 ( hex in ein Byte ASCI umwandeln )
  2 1 ?enough        dup 9 > if 7 + endif
  3                                    30 +
  4 ;
  5 : x@ ( n ==> addr                       )
  6 ( Adresse des n. Folgeblocks aus )
  7 ( dem aktuellen Menue                   )
  8 1 ?enough           1 - 3 * m@ block
  9       21+ + dup c@ cn 8 <shift swap
 10          1+ dup c@ cn 4 <shift swap
 11                        1+ c@ cn or or
 12 ;
 13 : t?  ( blockaddr ==> ctyp              )
 14 1 ?enough                       1f+ c@
 15 ; -->
scr # 11
  0 : x!  ( b n ==>                         )
  1 ( Block-Nummer b in ASCCI als n. )
  2 ( Parameter ins Menue eintragen      )
  3 2 ?enough
  4       1 - 3 * m@ block 21+ + addr !
  5 dup f00 and 8 >shift nc addr @   c!
  6  dup 0f0 and 4 >shift nc addr @ 1+
  7   c! 00f and nc addr @ 2+ c! update
  8 ;
  9 : x-.   ( blockaddr ==>                 )
 10 1 ?enough              1f -trailing type
 11 ;
 12 : xt?  ( b ==> addr typ                 )
 13 1 ?enough              block 1f+ dup c@
 14 ; -->
 15
scr # 12
  0 : header  ( n ==>                       )
  1 ( Block n wird Ueberschrift             )
  2 1 ?enough hex nok 93 emit dup 4 .r
  3   ."  ↑ (  " en @ 4 .r ." . Ebene "
  4          vn @ 5 .r ." . Aufruf )" cr
  5             22 emit block dup t? case
  6    3e of ."  Menue: " endof
  7    3d of ."  Antw.: " endof
  8    2d of ."  frei:  " endof endcase
  9                          1f type cr
 10 ;
 11 : scr?  ( b ==>                         )
 12 ( Block-Nummer als Screen-Zeile  )
 13 1 ?enough decimal dup 4 >shift cr
 14        . ."  edit" cr f and . cr
 15 ; -->
scr # 13
  0 : verify  ( ==>                         )
  1      graphic &hi-res vv fff 0 fill
  2         ev fff 0 fill bis anfang do
  3                          ff i ev + !
  4            i m! i block t? 3e = if
  5                      i m! 0b 1 do
  6             ff i x@ vv + c! loop
  7                                endif
  8                                  loop
  9                   bis anfang 1+ do
 10               vv i + dup c@ swap
 11                  0 swap c! 0= if
 12                       i x0 endif
 13                                  loop
 14                 &lo-res forth done
 15 ; -->
scr # 14
  0 : x.   ( n ==>                          )
  1 ( Menue-Zeile n ausgeben                )
  2 1 ?enough                22 emit space
  3      dup block dup 1f type t? case
  4       3e of ."  Menue" endof
  5       3d of ."  Antw." endof
  6       2d of ."  frei " endof endcase
  7           cr 21 spaces 4 .r ."  ↑" cr
  8 ;
  9 : prot  ( ==>                           )
 10 ( Protokoll                             )
 11                        cr vn @ 1+ 1 do
 12         vv i 4 * + dup @ swap 2+ @
 13          dup block swap 4 .r ."  "
 14              swap spaces  x-. cr loop
 15 ; -->
scr # 15
  0 : ↑  ( n ==>                            )
  1 ( Menue aus Block n ausgeben.           )
  2 1 ?enough     dup dup dup header m!
  3  en @ 1+ 7ff and dup en ! 2 * ev +
  4  ! vn @ 1+ 3ff and dup vn ! 4 * vv
  5     + dup en @ swap ! 2+ ! 0b 1 do
  6                    i x@ dup 0> if
  7            dup block t? 2d = if
  8                 0 i x! drop else
  9                        x. endif
 10                               else
 11                       drop endif
 12                                loop
 13 ;
 14 -e message cr -d message cr cr cr
 15 -->
scr # 16
  0 : lprot  ( ==>                          )
  1 ( Protokoll drucken                     )
  2                         cr vn @ 1+ 1 do
  3        vv i 4 * + dup @ swap 2+ @
  4  dup block prton swap 4 .r ."  "
  5         swap 2 * spaces x-. prtoff
  6       ?terminal if leave endif loop
  7 ;
  8 : ←  ( ==>                              )
  9 ( Ein Menue zurueckgehen                )
 10                           en @ 1 > if
 11                en @ 2 - dup en ! else
 12                      0 en ! 0 endif
 13                     1+ 2 * ev + @ ↑
 14 ; -->
 15
scr # 17
  0 : x0?  ( ==>                            )
  1 ( Freie Menue-Zeile suchen              )
  2                              0 0b 1 do
  3   i x@ 0= if drop i leave endif
  4                                    loop
  5    -dup 0= if -2 message cr endif
  6 ;
  7 : start  ( ==>                          )
  8                m@ dup vv ! dup ev !
  9                done 0 vn ! 0 en ! ↑
 10 ;
 11 : set  ( n ==> n                        )
 12 ( Verwaltung-Block aktualsieren  )
 13 1 ?enough        dup bis max ' bis !
 14                m@ bis 31 m! 3 x! m!
 15 ; -->
scr # 18
  0 : frei?  ( n ==> b                      )
  1 ( sucht einen freien Block ab       )
  2 ( Block n  b=0: kein Block              )
  3 1 ?enough        0 swap ende swap do
  4                i block t? 2d = if
  5                drop i leave endif
  6                                    loop
  7 ;
  8 : x?  ( ==> n                           )
  9 ( sucht freien Block                    )
 10                  bis frei? dup 0= if
 11      drop anfang frei? dup 0= if
 12             cr -3 message cr endif
 13                                  endif
 14    -->
 15
scr # 19
  0 : txt?  ( ==>                           )
  1 ( Text einlesen                         )
  2                 cr cr txt 1f 20 fill
  3 ." ________________________________________"
  4           cr 91 dup emit emit txt 20
  5     expect txt cr dup 20 + swap do
  6                           i c@ 0= if
  7                       20 i c! endif
  8                                    loop
  9 ;
 10 : x-pert  ( ==>                         )
 11 cr empty-buffers -10 message 31 m!
 12      1 x@ ' anfang ! 2 x@ ' ende !
 13       3 x@ ' bis ! anfang m! start
 14 ; -->
 15
scr # 20
  0 : replace  ( n ==>                      )
  1 ( Block in Menuezeile n ersetzen )
  2 1 ?enough         3e m@ block 1f+ c!
  3                          x? dup 0> if
  4     set dup rot x! txt? block dup
  5      txt swap 1f cmove 1f+ dup 3d
  6 swap c! 2+ 1e 20 fill update endif
```

```
  7 ;
  8 : x+  ( ==>
  9 ( Block an Menue anhaengen.                )
 10       x0? -dup 0> if replace endif
 11 ;
 12 : save  ( ==>
 13 ( Aenderungen sichern                      )
 14                                   done +
 15 ; -->

scr # 21
  0 : new-disk  ( ==>                          )
  1 ( Neue Daten-Diskette einrichten           )
  2            2 list 3 list 4 list 5 list
  3             31 m! 60 dup 1 x! dup 3 x!
  4                 dup ' anfang ! ' bis !
  5   a2f dup 2 x! ' ende ! 93 emit -8
  6  message cr -7 message cr key drop
  7                       " n0:x-pert,xp" dos
  8  create-screen 2 2 scopy 3 3 scopy
  9                     20 10 do i x0 loop
 10                                 a3  6 do
 11                          1 i scopy loop
 12         ." Titel eingeben " cr 31 m!
 13                  1 replace done x-pert
 14 ; -->
 15

scr # 22
  0 : print  ( ==>                             )
  1 ( Ausdruck aller Eintraege                 )
  2                       hex bis 1+ anfang do
  3              i m! i block t? 2d = if
  4           i 4 .r ."    frei" cr else
  5   prton i 4 .r space i block 1f
  6      type space i block t? emit
  7    space 0b 1 do i x@ 4 .r loop
  8                        prtoff endif
  9            ?terminal if leave endif
 10                                   loop
 11 ; -->
 12
 13
 14
 15

scr # 23
  0 : tree ( nb ==> nb                         )
  1   r> drop depth 0= ?terminal or if
  2                             quit endif
  3       dup f000 and c >shift 0b = if
  4                             drop else
  5       dup f000 and c >shift 0= if
  6          1000 + dup 0fff and dup
  7     block prton swap 4 .r depth
  8          2 * spaces x-. prtoff else
  9        1000 + dup 0fff and m! dup
 10    f000 and c >shift 1 - x@ dup
 11                 0= if drop endif
 12                                endif
 13                                endif
 14          [ smudge ] tree [ smudge ]
 15 ; -->

scr # 24
  0 : baum  ( ==>                              )
  1 ( Baum des aktuellen Menues                )
  2       prton prtoff clear m@ tree cr
  3 ;
  4 : x-  ( n ==>                              )
  5 ( Antwort-Block freigeben                  )
  6 1 ?enough            dup xt? 3d = if
  7                          drop x0 else
  8          cr -4 message cr 2drop endif
  9 ;
 10 : corr  ( n ==>                            )
 11 ( Text im Block veraendern                 )
 12 1 ?enough               txt? block txt
 13                  swap 1f cmove update
 14 ; -->
 15

scr # 25
  0 : x=  ( n ==>                              )
  1 ( Menue-Block in Antwort wandeln           )
  2 1 ?enough         dup m! xt? 3e = if
  3   0 0b 1 do i x@ + loop 0= if
  4        update 3d swap c! else
  5   cr -6 message cr drop endif else
  6          cr -5 message cr drop endif
  7 ;
  8 : x<  ( n ==>                              )
  9 ( Menue-Blk vor Zeile einfuegen            )
 10 1 ?enough   nach ! m@ vor ! 0b 1 do
 11                   i x@ nach @ = if
 12 i replace 3e i x@ dup m! block
 13   1f+ c! nach @ 1 x! leave endif
 14                   loop vor @ m!
 15 ; -->

scr # 26
  0 : x+^  ( n ==>                             )
  1 ( bestehendes Menue n anfuegen             )
  2 1 ?enough                    0 0b 1 do
  3                           i x@ 0= if
  4                drop i leave endif
  5                                   loop
  6                              dup 0= if
  7              -2 message 2drop else
  8     x! 3e men @ block 1f+ c! endif
  9 ;
 10
 11  39 block drop
 12  -9 message cr cr
 13  -7 message ."   " cr
 14 close-screen key drop x-pert
 15
```

Listing 1. »x-pert«, ein Mini-Expertensystem (Schluß)

```
scr # 40
  0 (       Korrektur von triad              )
  1 ( Danach arbeitet das Wort wie           )
  2 ( im Handbuch beschrieben                )
  3 ' nop           ( PFA von 'nop'          )
  4   cfa           ( in CFA wandeln         )
  5   dup           ( wird 2x benoetigt      )
  6 ' triad !       ( 1.                     )
  7 ' triad 2+ !    ( und 2. Wort von        )
  8                 ( 'TRIAD' mit 'NOP'      )
  9                 ( ueberschreiben         )
 10
 11
 12 a2f 1f30 ! ( 162 Screens ermoeg-         )
 13            ( lichen, Aenderung           )
 14            ( in r/w                      )
 15 -->

scr # 41
  0 (   Konstante 'dum' einrichten           )
  1
  2 : £ ( addr wert ==> addr+2 ) swap dup 2+ rot rot ! ;
  3 : ££
  4                  2c10 dup dup  ( Addr in dump )
  5           ' dump lfa @ = 0= if  ( dum nicht eingerichtet )
  6              4483 £ cd55 £     ( Namensfedld eintragen )
  7                dump lfa @ £    ( lf von dump uebernehmen )
  8                  ' 0 cfa @ £   ( Codefeld fuer Konstante )
  9                         10 £   ( Wert eintragen )
 10        drop ' dump lfa swap £  ( neues Linkfeld fuer dump )
 11                          endif
 12 ;
 13 ££ drop forget £  ( Hilfsworte loeschen )
 14 -->
 15
```

Listing 2. Zwei Verbesserungen zu Forth 64

```
scr # 42
  0 ( dump  von bis ==>  <neue Vers> )
  1 ( dump in der Breite dum mit '.' )
  2 ( als nicht druckbares Zeichen   )
  3 : £ 2 ?enough          1+ base @ hex
  4           -rot swap dup rot swap do
  5        nop cr i 0 5 d.r dum 0 do
  6               dup c@ dup 3 .r
  7             dup 7f and 20 < if
  8                drop 2e endif
  9              pad i + c! 1 + loop    ( Text aufbauen )
 10               space pad dum type    ( Zeilentext ausgeben )
 11 ?terminal if leave endif dum +loop
 12                   drop cr base !
 13 ; -->
 14
 15
```

```
scr # 43
  0 (  alte Ver. dump ueberschreiben )
  1
  2 ' £ cfa dup ( Parameter-Feld     )
  3 here swap - ( Laenge             )
  4 ' dump cfa  ( Zieladresse = pfa  )
  5             ( der alten Version  )
  6 swap cmove  ( eintragen          )
  7
  8 forget £    ( Zwischenversion    )
  9             ( loeschen           )
 10 -->
 11 ( !!!! nun geht dump wieder !!!! )
 12
 13
 14
 15
```

Listing 2. Zwei Verbesserungen zu Forth 64 (Schluß)

Am Anfang war das Wort..

Programmieren in Forth bedeutet, neue Wörter zu schaffen, indem bereits definierte Befehle zu immer komplexeren Gruppen zusammengefaßt werden.

Forth ist eine sehr leistungsfähige Sprache. Wer in Forth programmiert, der programmiert nicht; der schafft sich neue Wörter, wobei am Ende oft ein einziges Wort steht, das dann das fertige Programm repräsentiert.

Diese Art des Programmierens bietet gegenüber der herkömmlichen Unterprogramm-Technik erhebliche Vorteile:

- Jedes Wort ist quasi eine Spracherweiterung, das heißt das Computersystem wächst mit dem Programmierer. Je mehr man in Forth programmiert hat, desto häufiger kann man auf Befehle zurückgreifen, die man Monate zuvor geschaffen hat. Dadurch nimmt die Produktivität des Programmierers enorm zu.
- Es ist einfacher, einzelne Wörter auf ihre Richtigkeit zu überprüfen, als ein ganzes Programm. Forth-Programme sind also zuverlässiger und man verliert weniger Zeit bei der Fehlersuche (Debugging).
- Die Programmpflege gestaltet sich erheblich einfacher als in allen anderen Sprachen, da, um ein Programm neu anzupassen, meist nur ganz wenige grundlegende Wörter abzuändern sind. Das ist für all diejenigen wichtig, die zum Beispiel vorhaben, irgendwann auf einen anderen Computer umzusteigen, und Ihre Programmsammlung dann weiter verwenden wollen. Wer einmal versucht hat, Basic-Programme vom C 64 auf einen anderen Computer umzuschreiben, der weiß das zu würdigen.

Daß Forth schwer zu erlernen sei, ist nur ein Gerücht. Forth ist sehr ungewöhnlich, aber auch nicht schwieriger als Basic oder Pascal. Nur wer sich nicht von gewohnten Konzepten trennen kann, mag anfangs Schwierigkeiten haben.

Wie schafft sich der Forth-Programmierer nun seine neuen Wörter? Es gibt prinzipiell drei Möglichkeiten, die die Stichwörter Colon-Definition, Primitive und Compiler umreißen.

Die Colon-Definition

Die Colon-Definition ist die übliche Methode, Wörter zu bauen. Die meisten Programme bestehen aus kaum mehr als einer Reihe von solchen Definitionen.

Eine Colon-Definition besteht aus:
Anfang, Name, Definition, Ende.

Den Anfang kennzeichnet in Forth ein Doppelpunkt (engl. Colon, daher der Name dieser Definitionsart), darauf folgt der Name des zu definierenden Wortes. Die dann folgende Definition ist nichts weiter als eine Liste bereits bekannter Forth-Wörter, die vom System immer dann ausgeführt werden sollen, wenn der neue Name als Anweisung auftaucht. Das Ende der Colon-Definition aktiviert ein Semikolon.

Ein Beispiel:

```
: PETRA 5 + . ;
```

VLIST zeigt, daß PETRA nun ganz oben im Dictionary steht! PETRA addiert eine 5 und zeigt das Ergebnis an. Zum Beispiel ergibt

```
10 PETRA
```

nach Drücken der Return-Taste die Meldung

```
15 OK
```

Die eben durchgeführte Colon-Definition von PETRA repräsentiert natürlich kein besonders gutes Beispiel für sinnvolle Programmierung in Forth, denn normalerweise wird man als Namen für ein Wort immer eine aussagekünftige Bezeichnung wählen (in diesem Falle vielleicht »PLUSFUENF.«). Die Bezeichnung PETRA sollte nur zeigen, daß man in Forth bei der Namensgebung völlig freie Hand hat.

Forth und Maschinensprache

Wer etwas Ahnung von Maschinensprache hat, der sollte nicht denken, »da brauche ich Forth ja nicht mehr«, sondern kann seine Kenntnisse sehr sinnvoll einbringen. Es besteht die Möglichkeit, Wörter statt wie eben in »High-Level«, also als Colon-Definition, auch als sogenannte »Primitive«, das heißt in Maschinensprache, zu schreiben. Dazu benötigt man einen Forth-Assembler, den die meisten Forth-Versionen gleich mitanbieten.

Ein Maschinen-Befehl besteht in der Regel aus Befehlswort und Operanden.

Im 6502-Assembler lautet beispielsweise der Befehl zum Laden des Akkumulativs mit der Konstanten 1

```
LDA #1
```

Der Forth-Assembler benötigt hier die umgekehrte Notation, also erst den Operanden:

```
01 # LDA,
```

(man beachte das Komma hinter LDA!).

Als Beispiel ein kleiner Vergleich:

6502-Assembler

```
$C000 CLC                $C009 RTS
$C001 LDA $D020          $C00A NOP
$C004 ADC   #$01         $C00B NOP
$C006 STA $D020
```

Forth

```
SCR  #  1
 0           (PRIMITIVES)
 1
 2 CODE DANI          CLC,
 3           53280   LDA,
 4               1 # ADC,
 5           53280   STA,
 6            NEXT   JMP,  END-CODE
```

NEXT ist die Einsprungstelle für den »Inneren Interpreter« von Forth. Der Innere Interpreter kümmert sich darum, welches Wort als nächstes an die Reihe kommt, dem »Äußeren Interpreter« fällt die Aufgabe Benutzer und Eingaben zu.

Es ist empfehlenswert, Programme zunächst in High-Level zu erzeugen; falls die Geschwindigkeit dann nicht ausreicht, genügt es meistens, ein oder zwei Worte als Primitive umzuschreiben. Das ist erheblich effektiver, als würde man das ganze Programm in Assembler schreiben. Oft erübrigt sich auch das, da Forth sowieso 100- bis 400-mal schneller als Basic läuft.

Und nun für ganz Raffinierte: Wir können Wörter erfinden, die uns die »Arbeit« des Wörterbauens abnehmen. Dieser Punkt macht Forth so leistungsfähig. Als typischer Vertreter dieser Wörter steht CONSTANT. »n CONSTANT name« schafft ein neues Wort namens »name«, das bei Aufruf n auf den Stack legt. Beispiel:

```
1024 CONSTANT SCREEN
```

definiert zunächst das Wort SCREEN. Der Leser kann sich davon mit Hilfe von VLIST überzeugen.

SCREEN legt dann den Wert 1024 auf den Stapel. Beispiel:

```
SCREEN 500 + .
 1524 OK
```

Anstelle von 1024 (Adresse des ersten Byte des Bildschirmspeichers) kann also überall im Programm SCREEN einspringen.

Will man später das Programm auf einem Computer laufen lassen, bei dem der Bildschirmspeicher vielleicht bei 4096 beginnt, so braucht man nur ein einziges Wort abändern:

```
: SCREEN 4096 ;
```

Praktisch, nicht wahr?

Das Wort CONSTANT schafft also eine ganz neue Klasse von Wörtern, eben mit der Eigenschaft, Konstanten zu sein.

Ein Wort wie CONSTANT muß also zweierlei tun: Erstens das Wort »name« ins Dictionary eintragen, und zweitens »name« sagen, was es tun soll, wenn der Benutzer »name« aufruft.

Zunächst erfinden wir also mit Hilfe einer Colon-Definition ein Wort wie CONSTANT. Nennen wir es ANGEBER. ANGEBER schafft nun eine neue Klasse von Wörtern, nämlich Angeber-Wörter. Das sieht so aus:

```
: ANGEBER <BUILDS DOES> ;
```

Zwischen <BUILDS und DOES> steht die Angabe, wie der Wortkörper auszusehen hat, hier steht also gar nichts; zwischen DOES> und »;« steht, was die Angeber-Wörter dann ausführen sollen, hier also ebenfalls nichts. DOES> bewirkt, daß die PFA (Parameterfeldadresse) des Wortes »name« auf den Stack gelegt wird. Die PFA ist die Adresse des ersten Bytes des Parameterfeldes. Das Parameterfeld beinhaltet eine Liste im Wortkörper, die das Wort zur Ausführung benötigt. Bei Primitives steht da der ganze Maschinencode, bei High-Level-Wörtern besteht das Parameterfeld aus einer Liste der Adressen der Wörter, die abgearbeitet werden sollen, die also in der Colon-Definition standen.

Probieren wir ANGEBER einmal aus:

```
ANGEBER MICHAEL
```

Wir überzeugen uns mit VLIST, daß Michael als neues Wort im Dictionary steht! Nun rufen wir MICHAEL auf. Was passiert? Nichts, außer daß die PFA von MICHAEL auf dem Stack liegt. Deshalb ist MICHAEL ein Angeber: er gibt seine PFA an, aber nichts dahinter!

Probieren wir etwas anderes:

```
: NICHTSNUTZ <BUILDS DOES> DROP ;
```

Wir vermuten, die Nichtsnutz-Wörter entfernen sogar noch ihre PFA. Wir geben zunächst ein:

```
NICHTSNUTZ WALTER
```

und dann

```
WALTER
```

und tatsächlich, nichts passiert.

Versuchen wir einmal, Konstanten bauen zu lassen:

```
: CONSTANTINOPEL <BUILDS , DOES> @ ;
```

Das »,« bewirkt, daß die oberste Zahl vom Stack genommen und ins Dictionary, also in das Parameterfeld, gepackt wird. Die durch CONSTANTINOPEL definierten Wörter holen diese Zahl wegen @dann wieder auf den Stack. Der Vorgang wiederholt sich dauernd. Beispiel:

```
1024 CONSTANTINOPEL SCREEN
SCREEN 500 + .
1524 OK
```

Aha! Prima, aber wir hätten es natürlich auch einfacher haben können:

```
: CONSTANTINOPEL CONSTANT ;
```

Zum Abschluß wollen wir noch ein »richtiges« Programm schreiben, nämlich einen Maskengenerator, der die Bildschirmseite irgendwo speichert und bei Bedarf immer wieder holen kann. Am besten bringen wir den ganzen Bildschirm in einer einzigen Variablen unter. Der Leser wird schon erraten haben: Wir benötigen einen neuen Variablentyp. Nennen wir ihn MASKE. Die Masken sollen in ihrem Parameterfeld zunächst 1000 Leerzeichen, also 1000mal den Bildschirmcode 32 haben. Das erledigt eine Schleife, also in Basic:

```
FOR I=0 TO 999 ......NEXT
```

Das sieht in Forth etwas anders aus:

```
1000 0 DO ......LOOP
```

Die Punkte stehen für eine Anweisung. Nun brauchen wir noch ein Wort, um die 1000 Byte zu transportieren. Dazu bietet uns Forth

```
CMOVE ( a  a  n - )
```

an. CMOVE verlagert n-Byte von der Adresse a nach a.

PFA 1024 1000 CMOVE bringt uns also die Bytes aus der Maske (ab a PFA) in den Bildschirmspeicher.

Den Ausdruck

```
1024 1000 CMOVE
```

kürzen wir durch »M@« ab.

```
MASKE ANDREA
```

kreiert uns eine Leermaske namens ANDREA.

```
ANDREA M@
```

holt uns ANDREA in den Bildschirmspeicher. Zum Speichern mit »M!« werden nur die Adressen a und a vertauscht (siehe Listing). Damit besitzen wir bereits einen primitiven Maskengenerator! Wir können ihn als »Notizzettelspeicher« verwenden, oder als »virtual screen buffer«, oder um mit mehreren Bildschirmen zu arbeiten, und so weiter.

MASKMAKER ist eine etwas komfortablere Version: PAGE löscht den Bildschirm (ASCII-Code 147 beim C 64), KEY erwartet solange eine Tastatureingabe, die mit EMIT auf den Bildschirm gebracht, bis RETURN gedrückt wird. Dann wird die fertige Maske mit »M!« gespeichert.

Wer will jetzt noch behaupten, Forth sei unübersichtlich? Man beachte, daß der ganze Maskengenerator lediglich 127 Byte lang ist, also gerade zwei oder drei Basic-Zeilen entspricht.

(Andreas Carl/ev)